EN DATA BECKER BOG

DITTRICH

# COMMODORE AMIGA

# MASKINSPROG

Maskinsprog gør livet lettere!

DANSK/NORSK udgave

Forlag: NCS, box 105, DK 6950 Ringkøbing

# ADMINISTRATIVE PROGRAMMER TIL PC

**Finans-bogholderi**
**Debitor- og kreditor-bogholderi**
**Fakturering**
**Lagerstyring**

**NORDIC COMPUTER SOFTWARE**
Kundeservice kl. 09.00 - 10.00, tlf. 07 32 02 02

# MASKINSPROG

***EN DATA BECKER BOG***

DITTRICH

# COMMODORE

# AMIGA

# MASKINSPROG

Maskinsprog gør livet lettere!

DANSK NORSK udgave

NORDIC COMPUTER SOFTWARE
SMEDEGADE 7 . BOX 105 . DK-6950 RINGKØBING
1987

KUNDESERVICE:
TLF. 07 32 02 02, KL. 09.00-10.00

Copyright 1987 Data Becker & Nordic Computer Software
Postbox 105
Smedegade 7
DK-6950 Ringkøbing

Sats & tryk: Tarm Bogtryk & Offset A/S

Dansk oversættelse & bearbejdning: E. J. Fjerdingstad

ISBN 87-7283-023-9

Alle rettigheder til den danske version tilhører Nordic Computer Software. Bogen må under ingen form (fotokopi, aftryk el. lign) reproduceres uden skriftlig tilladelse fra udgiveren. Ligeledes må bogen, eller dele heraf, ikke udbredes via elektroniske medier.

## VIGTIGT

Programmer, programeksempler mv. er omfattet af lov om copyright. Disse må kun anvendes til personlige- eller undervisningsformål og må ikke anvendes kommercielt.

Alle programeksempler, tekniske anvisninger o.s.v. i denne bog er omhyggeligt gennemgået for fejl, ligesom der er læst korrektur mindst 3 gange på oversættelsen af teksten.
Alligevel forekommer enkelte fejl, og skulle en læser falde over en sådan, vil vi være meget taknemlige for en skriftlig meddelelse herom, så evt. fremtidige udgaver kan blive så meget bedre.

# Vigtig henvisning

*Alle koblinger, metoder og programmer i denne bog gengives uden hensyntagen til patentrettigheder. De er udelukkende beregnet på amatør- og undervisningsformål og må ikke benyttes til erhvervsformål.*

Samtlige koblinger, tekniske angivelser og programmer er udarbejdet, eller sammenstillet, af forfatterne med den største omhyggelighed og reproduceret under anvendelse af effektive kontrolforanstaltninger. Alligevel kan fejl ikke udelukkes. DATA-BECKER og NCS ser sig derfor nødsaget til at bemærke, at det ikke kan påtage sig nogen garanti eller juridisk ansvar, eller anden hæftelse for følger af fejlagtige oplysninger. Forlaget er til enhver tid taknemlig for skriftlige oplysninger om eventuelle fejl.

# Forord

Det er altid forfriskende at få en god faglitteratur i hånden. Løser bogen så desuden et akut problem; kan man da næsten ikke ønske mere. Og det var netop den følelse »AMIGA maskinsprog« gav mig, da jeg første gang stiftede bekendtskab med den tyske udgave.

Når man dagligt igennem lang tid arbejder intens med den omfattende litteratur, der findes til Commodore-AMIGA, så kan man kun glæde sig over den gennemførte dokumentation, der findes for C-programmører. Og samtidig ærgre sig over den næsten totale mangel på dokumentation for MC 68000-maskinsprogs programmører.

Sådan dokumentation modtages derfor altid med glæde, og en god portion skepsis, for det er erfaringsmæssigt et meget svært stof at formidle på en let måde.

Selvom stofområdet er meget stort, synes jeg at »AMIGA maskinsprog« har formået at formidle det på en yderst tilgængelig og let-fordøjelig måde, og jeg anbefaler gerne enhver maskinsprogs-interesseret programmør at gå igang. Bogens eksempler er let forståelige, de virker, og frem for alt: de er eksempler på ting enhver AMIGA-programmør har brug for.

Denne danske udgave bør stå hos enhver AMIGA-programmør!

God fornøjelse!

Jan Nymand
Produktchef
Commodore Data A/S

# Forfatterens forord

Commodore Amiga betragtes med rette som en meget alsidig maskine, hvis ydeevne opfordrer til at anvende den i egne programmer.

Der findes allerede en række programmeringssprog, hvori Amigaen programmeres. Et af dem, hvormed det hele begynder: maskinsproget. Det er i det, at den første kompilator til denne computer. Visse dele af operativsystemet er også skrevet i maskinsprog.

For at programmere computeren, skal man ikke alene vide noget om processoren MC68000 og dens forråd af ordrer, men også om interne forhold i Amigaen selv. De mange funktioner, som denne computers operativsystem byder på, skal nemlig kunne kaldes og betjenes af et maskinsprogsprogram.

Det udgjorde hidtil et vist problem. Dokumentationen om Amiga er næsten udelukkende tænkt for C-programmører, og er således ikke nogen stor hjælp ved maskinsprogsprogrammering. Da jeg på en højere lærenanstalt i Køln skulle udvikle en emulator til 6502 processoren på Amigaen, måtte jeg kæmpe med dette problem. Efter lange dage i laboratoriet og hjemme ved computeren, kan jeg således forelægge Dem en art ny dokumentation, der beskriver processoren i Amiga og dens tilgange til operativsystemet.

Først gennemgås nogle grundlæggende træk som lagerorganisation og computerens "lille tabel", hvorefter der følger et kig i det indre af Amigaen, og en beskrivelse af processoren. Ved hjælp af nogle programeksempler, kan man bagefter programmere processoren, og lære mere herigennem.

De skal lære, hvor let man med sådanne programmer kan lokke så smukke ting, som vinduer og menuer ud af Amigaen, eller indlæse data og tekster, og igen få dem udlæst. Alle disse funktioner illustreres med programeksempler, som man kan samle og på den måde erhverve et bibliotek af brugbare delprogrammer. Ved hjælp af dette bibliotek, kan man producere hurtige, brugervenlige og professionelt udseende programmer.

Jeg ønsker Dem megen fornøjelse med denne bog og Amigaen under efterprøvningen og anvendelsen af den viden, som De snart vil have erhvervet.

*Stefan Dittrich* *Gummersbach, april 1987*

# Indholdsfortegnelse


1. Indledning ........ 11

1.1. Hvorfor maskinsprog? ........ 11
1.2. Et blik på lageret ........ 11
1.2.1. RAM, ROM og hardware-registre ........ 12
1.2.2. Bits, bytes og ord ........ 13
1.2.3. Talsystemer ........ 14
1.3. Amigas opbygning ........ 16
1.3.1. Struktur ........ 16
1.3.2. Lager-opdeling ........ 17
1.3.3. Alting på een gang: multitasking ........ 18

2. Processoren MC68000 ........ 20

2.1. Registre ........ 20
2.2. Lager-adressering ........ 22
2.3. Funktionstilstande ........ 29
2.3.1. User- og supervisor-funktionsmåde ........ 29
2.3.2. Exceptions ........ 30
2.3.3. Interrupts ........ 33
2.3.4. Betingelsescodes ........ 34
2.4. Ordresættet ........ 37

3. Hvordan man arbejder med assemblere ........ 49

3.1. Udvidelses-programpakkens assembler ........ 49
3.2. Profimat Amiga ........ 51
3.3. K-SEKA assembleren ........ 52

4. De første programmer ........ 61

4.1. Addition af tal ........ 61
4.2. Sortering af en tabel ........ 64
4.3. Forvandling af talsystemer ........ 67
4.3.1. Binære til hexadecimale tal ........ 67
4.3.2. Binære til decimale tal ........ 70
4.3.3. Hexadecimale til binære tal ........ 71
4.3.4. Decimaltal til binære ........ 74

5. Hardware-register anvendelse .......... 76

5.1. Aflæsning af specialtaster .......... 76
5.2. Hvordan man fastslår den løbende tid .......... 77
5.3. Hvordan man udlæser mus og joystick .......... 78
5.4. Betjening af parallel-interface't .......... 80
5.5. Tonefrembringelse .......... 83
5.6. Oversigt over hardware-registrene .......... 89

6. Hvordan man arbejder med Amiga-operativsystemet .......... 92

6.1. Hvordan man load'er biblioteker .......... 92
6.2. Hvordan man kalder funktioner .......... 94
6.3. Programforberedelse .......... 95
6.3.1. Rservering af lagerplads .......... 96
6.3.2. Åbning af et vindue på nemmeste måde .......... 97
6.4. Ind- og udlæsning .......... 101
6.4.1. Skærmbilledudlæsninger .......... 101
6.4.2. Indlæsning med tastaturet .......... 106
6.4.3. Printer-tilgang .......... 109
6.4.4. Seriel ind-/udlæsning .......... 109
6.4.5. Sprogudlæsning .......... 110
6.5. Disketteoperationer .......... 115
6.5.1. Åbning af filer .......... 116
6.5.2. Læsning og skrivning af filer .......... 117
6.5.3. Sletning af filer .......... 119
6.5.4. Ombenævnelse af filer .......... 120
6.5.5. Udlæsning af indholdsfortegnelse .......... 120
6.5.6. Direkte tilgang til disketten .......... 125

7. Arbejdet med Intuition .......... 131

7.1. Åbning af skærmbillede .......... 132
7.2. Åbning af vinduer .......... 142
7.3. Meldinger og aflæsninger: Requester .......... 147
7.4. Behandling af hændelser .......... 150
7.5. Menu-programmering .......... 152
7.6. Tekstudlæsning .......... 166
7.7. Images .......... 168
7.8. Borders .......... 171
7.9. Gadgets .......... 174
7.9.1. Boolean-gadgets .......... 174
7.9.2. Streng-gadgets .......... 181
7.9.3. Proportional-gadgets .......... 186

8. Programeksempler ........ 191

8.1. Omstilling til supervisor-modus ........ 191
8.2. Exception-programmering ........ 192

Appendix ........ 196

Ordforklaring ........ 210

# 1. Indledning

Før vi kaster os over arbejdet med maskinsprog, bør vi se på nogle ting, som er vigtige for forståelsen af maskinsprogsprogrammering. Det bør også nævnes, hvad der netop er så interessant ved dette sprog.

## 1.1. Hvorfor maskinsprog?

Maskinsprog er egentlig det eneste sprog, som processoren MC68000 forstår. Alle andre sprog, som det højtelskede BASIC eller C, skal derfor oversættes til maskinsprog, hvad enten det sker under programudførelsen (som med en fortolker, f.eks. til BASIC) eller tidligere (som med kompilator, f.eks. til C).

Det viser en fordel ved maskinsprog i forhold til de nævnte højere programmeringssprog: det er hurtigere. Ved anvendelse af en fortolker, skal hver enkelt linie hele tiden oversættes på ny, hvad der kræver tid. Og den oversættelse, som en kompilator leverer, er, når alt kommer til alt, selv et makinsprogsprogram, der, på grund af den generelle oversættelsesmetode, aldrig kan være så optimeret, som et rent maskinsprogsprogram.

Hertil kommer, at man ikke behøver nogen fortolker til udførelse af et rent maskinsprogsprogram, hvad der er en yderligere fordel i forhold til BASIC.

Med maskinsprog Kan man bedre udnytte alle muligheder ved en computer, hvad der især lønner sig ved en så mangesidig computer som Amigaen. Således kan det undertiden være nødvendigt, at tilføje et maskinsprogsprogram til et program i et højere programmeringssprog, hvis man ikke kan tilgå en eller anden funktion i computeren med det pågældende programmeringssprog.

Til sidst vil jeg nævne et argument til: for rigtige assemblerentusiaster (som mig), der er øvet i assembler-programmering fra en anden computers 6502 eller 68000, er der slet ingen diskussion om, hvilket programmeringssprog, der skal anvendes.

## 1.2. Et blik på lageret

Før man vil skrive et program i maskinsprog, må man gøre sig klart, hvad dette program skal gøre, og hvilke hjælpemidler, der kræves til det. Det vigtigste hjælpemiddel er vel det lager, som der arbejdes med. Det er uundgåeligt, at skulle lagre data og programmer.

Et sådant lager er en mangesidig sag. Mulighederne for dets anvendelse overstiger tydeligvis den for BASIC-programmører almindelige indstilling ”enten slår det til, eller også gør det ikke”, da der er nogle variationsmuligheder.

## 1.2.1. RAM, ROM og hardware-registre

Det første, man tænker på ved ordet ”lager”, er, at man kan indlæse data, og få dem ud igen. Denne lagringsmetode kaldes RAM, Random Access Memory, hvad der frit oversat betyder ”lager med direkte tilgang”. Lageret består af elektroniske komponenter, der kan opbevare data, der er indlæst i dem. Manglen er ganske vist, at alle data er væk, når computeren slukkes, eller strømmen svigter.

For at kunne komme igang, når strømmen sættes til, som f.eks. en opfordring til indlægge ”bootstrap”-disketten, skal der være et program til det i lageret. Det kan ikke ligge i RAM'en, vi har altså brug for endnu en type mere permanent lager: en ROM.

ROM betyder ”Read Only Memory”, og viser således, at den kun kan læses, men ikke skrives på. Sådan en ROM er indbygget i Amigaen, og sørger for load'ningen af kickstart-disketten ind i RAM'en. På disketten afgør programmerne, hvad der videre skal ske. I senere versioner af Amigaen findes kickstart-programmet i ROM'en, så computeren med det samme ved, hvad den skal gøre med Workbench-disketten.

En variant af ROM'er er de såkaldte PROM'er, ”Programmable Read Only Memory”-komponenter. Det er ROM'er, som man med særlige metoder kan programmere eller skrive på. De er ganske vist ret sjældne, fordi man ikke kan slette, hvad der een gang er skrevet på dem. Hyppigere end PROM'er er EPROM'er, ”Erasable Programmable ROMs”, der kan slettes ved bestråling med ultraviolet lys. De kendes på, at der er et lille vindue i chip'en.

Der findes desværre ikke i den normale handel, og ikke slet så billige som RAM-komponenter, såkaldte EEROM'er, ”Electrically Erasable ROMs”. Komponenterne kan slettes elektrisk, og ligner heri RAM-komponenter, men de taber ikke deres indhold, når strømmen afbrydes.

Efter at Amiga 500 er kommet frem, findes endnu en art lager, WOM'en. Ikke den i aprilnummeret af et elektronik-tidsskrift omtalte ”Write Only Memory”, der ikke er særlig anvendelig, men en ”Write Once Memory”. I et lager af denne type indlæses kickstart-disketten, og så kan der ikke længere skrives i det. Ganske vist er det ikke en ny art komponent, men et RAM-

område. Efter "boot"-ningen bliver den nødvendige signalledning til skrivning på denne RAM spærret, så der herefter kun kan udlæses fra lageret.

Foruden RAM'erne og de forskellige typer ROM'er findes endnu en lagertype, som ligger et sted imellem dem. Det drejer sig om lagerceller, der står i direkte forbindelse med perifere komponenter rundt omkring processoren. Disse lagerceller kalder man hardware-registre, fordi de styrer hardwaren. Hvordan man kan styre disse registre, kan De se i kapitlet "Hardware-register-anvendelse".

Men tilbage til det "normale" lager, RAM'en. Lad os engang betragte dets opbygning og anvendelse.

## 1.2.2. Bits, bytes og ord

*Den almindelige enhed, som man måler eller angiver lagerplads i, er kilobytes (Kbytes). En kilobyte er 1024 bytes, ikke 1000, som man umiddelbart ville antage. Denne ejendommelighed skyldes computerens binært orienterede opbygning, hvor de fleste tal angives som potenser af 2. Således også kilobytes.*

For at kunne tilgå et lager på 1 Kbyte kræver processoren 10 adresseledninger, hvori der enten ligger 5 volt eller 0 volt. Således opnår man 2↑10 = 1024 forskellige kombinationer og dermed lagerbytes.

En byte består af 8 enkelte ja/nej-informationer, så man i en byte kan lagre 2↑8 = 256 forskellige tal mellem 0 og 255. Disse enkelte informationsenheder, der er de mindste enheder i datalogien, kaldes bits. Det er en forkortelse for "Binary coded digit".

I et lager på 512 Kbytes, som i Amigaen, er der altså 2↑19 = 524288 bytes, dvs. 4194304 bits. Man skal tænke på, at man på den måde har 2↑4194304 forskellige kombinationsmuligheder, hvorved det turde være klart, at der hele tiden kan komme nye programmer...

Tilbage til enhederne. Bit og bytes er helt tilstrækkelige til at programmere en 8-bit processor som 6502'eren, da den kun kan arbejde med bytes. Ved den 16/32-bits processor, MC68000, der er indbygget i Amigaen, må man kende to datatyper til. Hermed menes ord, der består af 16 bits eller 2 bytes og de "lange ord", der er på 32 bits eller 4 bytes, eller 2 ord.

I et ord kan man lagre et tal mellem 0 og 65535, i et langt ord endda mellem 0 og 4294967295. Med så kolossale tal kan MC68000 nemlig arbejde, altså forskyde eller beregne, med en enkelt kommando.

Af og til behøver man også negative værdier ved præsentation af tal. Da en bit kun kan være 0 eller 1 og ikke -1, har man en vedtægt. Et ord, der skal behandles med fortegn, bærer i sin mest betydende bit, den femtende (man begynder at tælle ved nul), informationen om sit fortegn. På den måde kan der med t ord repræsenteres værdier mellem -32768 og +32767. For en byte bliver området fra -128 til +127. Værdien -1 bliver i bytes repræsenteret af $FF og i ord af $FFFF osv.

Lad os blive ved de positive værdier. For at udforme repræsentationen af en byte lettere i forhold til dens bitmønster, anvender man i maskinsprog, hvor dette bitmønster undertiden er meget vigtigt, ikke længere det kendte titalssystem. Derimod er det almindeligt med det binære, det oktale og det hexadecimale talsystem.

## 1.2.3. Talsystemer

Lad os se på titalsystemet, også kaldet det decimale talsystem. I systemet er grundtallet 10, dvs., at hver plads i et tal repræsenterer en potens af 10. Således betyder tallet 246 2*10↑2+4*10↑1+6*10↑0. Tegnforrådet består af 10 tegn, nemlig 0 til 9.

Ganske anderledes er det binære talsystem. Dette systems tegnforråd er 2, nemlig 0 og 1. Således er grundtallet her 2. Det binære tal 1010 betyder altså:

**1*2↑3+0*2↑2+1*2↑1+0*2↑0 = 2↑3+2↑1 = 8+2 = 10**
**(i titalsystemet)**

Binære tal kendetegnes i reglen med et %-tegn. Prøv at skrive titalsværdien af tallet %110011.

Nå, fik De også 51 ud af det? Det stemmer. Den nemmeste metode at foretage denne omregning på er, hver gang der er et ettal i tallet, at addere pladsværdierne. Pladsværdierne for de første 8 pladser er:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Plads:** | **8** | **7** | **6** | **5** | **4** | **3** | **2** | **1** |
| **Værdi:** | **128** | **64** | **32** | **16** | **8** | **4** | **2** | **1** |

Noget lignende gælder ved oktalsystemet. Dets grundtal er 8, tegnforrådet 0 til 7. Tallet 31 i oktalsystemet har således titalsværdien 3*8↑1+1*8↑0 = 25. Dette system er dog ikke så vigtigt, som det følgende:

Hexadecimalsystemet har 16 som grundtal. Tegnforrådet er 0 til F; altså er 10 i titalsystemet $A, hvor $-tegnet henviser til hexadecimalsystemet. Dette talsystem, er ved siden af det binære, det vigtigeste for assembler-programmeringen.

En byte, der kan indeholde tallene 0-255 repræsenteres altid med to pladser i det hexadecimale system: $00 til $FF. Et ord ligger mellem $0000 og $FFFF, et langt ord mellem $00000000 og $FFFFFFFF.

Omregningen af et binært tal til et hexadecimalt, der også kaldes et hextal, er meget let: man deler det binære tal op i grupper på fire. Hver af disse firegrupper repræsenterer en plads i det hexadecimale tal. Et eksempel:

| | | | |
|---|---|---|---|
| **Binært tal:** | **%110011101111** | | |
| **opdeles i:** | **%1100** | **%1110** | **%1111** |
| **der giver:** | **$C** | **$E** | **$F** |
| **altså:** | **%110011101111=$CEF** | | |

Omregningen i den modsatte retning er lige så let:

| | | | | |
|---|---|---|---|---|
| **Hextal:** | **$E30D** | | | |
| **opdeles i:** | **$E** | **$3** | **$0** | **$D** |
| **der giver:** | **%1110** | **%0011** | **%0000** | **%1101** |
| **altså:** | **$E30D=%1110001100001101** | | | |

Sådan kan man også gå frem med oktale tal, blot skal man her opdele det binære tal i pakker på tre:

| | | | | |
|---|---|---|---|---|
| **Oktaltal:** | **7531** | | | |
| **opdeles i:** | **7** | **5** | **3** | **1** |
| **der giver:** | **%111** | **%101** | **%011** | **%001** |
| **altså:** | **oktal 7531=111101011001** | | | |

Fra den binære form kan man, igen bringe det i den hexadecimale form:

| | | | |
|---|---|---|---|
| **Binært tal:** | **%111101011001** | | |
| **opdeles i:** | **%1111** | **0101** | **1001** |
| **der giver:** | **$F** | **$5** | **$9** |
| **altså:** | **oktalt 7531 = $F59** | | |

For at ændre tallet til titalsystemet, er det tilstrækkeligt med følgende omregning:

| | | | |
|---|---|---|---|
| **Hextal:** | **$F59** | | |
| **opdeles i:** | **$F** | **$5** | **$9** |
| **der giver:** | **15*16↑2 +5*16 +9** | | |
| **altså:** | **$F59 = 3929 decimalt.** | | |

Disse forvandlinger er ofte uundgåelige og alligevel besværlige. Med SEKA-assembleren er de nemme at lave, idet man indlæser værdien med "?" og får værdien skrevet ud i hexadecimal og decimal skrivemåde.

Tit skal forvandlingen ske indenfor et program, når man indlæser et tal i et kørende program, og dette tal skal videreforarbejdes. Det indtastede tal foreligger som en række tegn, der skal vurderes og normalt omdannes til et binært tal. Altså et ord, resp et langt ord, i lageret.
Det kan være nødvendigt, at forvandle tal den anden vej rundt. Hvis værdier, der beregnes i et program, skal udlæses i et eller andet talsystem, så skal værdien omregnes til en rækkefølge af tegn.

Til begge problemer vil vi i et senere kapitel udvikle maskinsprogsprogrammer, som man kan indsætte i egne programmer. Først skal vi se på nogle grundlæggende forhold, der er nødvendige for programmeringen.

## 1.3. Amigas opbygning

For at kunne beherske maskinsprog, er det ikke nok 'kun' at kende processoren og dens kommandoforråd. Man skal kende selve computeren lige så godt. Lad os engang betragte Amigaens indre.

### 1.3.1. Struktur

Amigaen er en meget ydedygtig maskine. Det skyldes, at der foruden processoren er indbygget nogle andre sværvægtere. Hermed mener jeg de såkladte custom-chips, der klarer adskillige opgaver selvstændigt.

*Det drejer sig først og fremmest om tre chips, der har de poetiske navne Agnus, Denise og Paula. Agnus, der også betegnes som en "blitter", er hovedsagelig beskæftiget med at forskyde lagerindhold, hvilket er nyttigt til hurtig bearbejdelse af billeder. Denise sørger for, at vi kan se noget på monitoren, og Paula er ansvarlig for ind-/udlæsninger (disketteoperationer eller lyd).*

Disse chips bliver betjent af processoren, via nogle adresser, der ligger fra $DFF000, og betegnes som hardware-registre (se også kapitlet herom i denne bog). For at lette den ret komplicerede betjening af disse chips, findes både i kickstart- og Workbench-disketten nogle biblioteker, der indeholder færdige service-programmer. Programmerne kan kaldes via mere eller mindre simple programdele, og overtager betjeningen af de tilsvarende chips.

Hvis man udelukkende anvender disse biblioteksfunktioner i sine programmer, så er Amigaens programmering i forskellige sprog ret ens. Om det drejer sig om assembler, BASIC eller C, så er de parametre, der skal overgives til funktionerne, de samme, kun skrivemåden er forskellig. BASIC skal tages med et vist forbehold, da visse kommandoer i dette sprog oversættes af fortolkeren til de tilsvarende funktionskald, hvorfor man til BASIC-programmering egentlig ikke behøver at vide, hvordan Amiga udfører den ene eller den anden funktion.

Vi Fortsætter på det laveste niveau, maskinsprogets. De funktioner, som bibliotekerne indeholder, er maskinprogrammer og derfor i nær familie med vore programmer. Derfor kan vi helt undlade at anvende bibliotekerne, og gøre det hele selv, hvad der dog er så besværligt, at vi ikke vil forsøge det for enhver pris.

## 1.3.2. Lager-opdeling

Hvis vi går ud fra en Amiga 1000, bør vi se lidt på lageret. I normalversionen råder denne computer over 512 Kbytes RAM, der optager adresserne 00000 til $7FFFF, altså 0-524287. Øger man lageret til en megabyte, så ligger de ekstra 512 Kbytes ikke lige direkte efter, men fra $200000 til $9FFFFF. Det område adskiller sig noget fra det lavere.

De chips i Amigaen, der understøtter processoren, har næsten alle selv direkte tilgang til lageret, og aflaster således processoren. Der er imidlertid en begrænsning. Disse chips kan kun benytte de første 512 Kbytes. Derfor kan de billed- og tonedata, der skal være tilgængelige for chipsene, kun ligge i den del af lageret, som man derfor også kalder chip-RAM.

Modsætningen til chip-RAMen er resten af RAM-området, der, hvis det er indbygget, ligger over $200000. Da processoren kan tilgå området alene, uden at blive "forstyrret" af de andre chips, kaldes det lager for fast-RAM (hurtig-RAM).

Her en oversigt over lagertildelingen i Amiga:

| | |
|---|---|
| **$000000- $07FFFF** | chip-RAM |
| **$080000- $1FFFFF** | reserveret |
| **$200000- $9FFFFF** | mulig fast-RAM |
| **$A00000- $BEFFFF** | reserveret |
| **$BFD000-$BFDF00** | PIA B (lige adresser) |
| **$BFE001-$BFEF01** | PIA A (ulige adresser) |

| | |
|---|---|
| **$C00000- $DFEFFF** | reserveret til senere udvidelser |
| **$DFF000- $DFFFFF** | hardware-register for custom-chips |
| **$E00000- $E7FFFF** | reserveret |
| **$E80000- $EFFFFF** | adresseområde for udvidelsesporten |
| **$F00000- $F7FFFF** | reserveret |
| **$F80000- $FFFFFF** | system-ROM |

Når vi load'er et program ind i lageret, bliver det læsset af et eller andet sted. Den lagerplads, der således bliver optaget, bliver noteret i en reserveringsliste, og er derefter spærret for anden anvendelse. Hvis man load'er et program til, hvad uden videre er muligt på Amigaen, så bliver det anbragt et andet sted, og optager yderligere lagerplads. Hvis det første program behøver mere plads, f.eks. til tekster, må det reservere sig plads. Ellers kan det være, at lagerområdet, og dermed teksterne, bliver overskrevet af et følgende program.

Herunder er det interessant, at man nu kan afslutte det første program og således atter frigive lagerplads. Så er lageret opdelt i frie og optagede dele, der ikke længere hænger sammen. Alligevel kan Amigaen stadig arbejde med disse lagerstumper, som om de var et sammenhængende stykke! Det viser sig særligt med den dynamiske RAM-disk, der altid er til rådighed under navnet RAM:.

*RAM-disken er egentlig et fænomen, da den altid er 100% optaget. Hvis et program slettes fra RAM-disk'en, så bliver den lagerplads, det optog, givet fuldstændigt tilbage til systemet, ligegyldigt hvordan og hvor den var fordelt. Hvis man altså reserverer og fylder et lagerområde på f.eks. 100 Kbytes, kan det godt ske, at det i virkeligheden ikke ligger i sammenhæng i lageret. Det mærker programmøren ikke noget til, da det alligevel kan håndteres, som om det var sammenhængende. Forskellen mellem det tilsyneladende og fysiske lager udlignes automatisk af Amigaen.*

## 1.3.3. Alting på een gang: multitasking

Amigaen er en underlig kasse, den kan gøre flere ting på n gang. I det ene vindue hopper en rød bold frem og tilbage, medens man kan bearbejder tekst i et andet vindue, og samtidig ser et ur tikke i hjørnet.

Det er i hvert fald det indtryk, man får, når man får demonstreret Amigaen. Der er ganske vist en hage ved sagen: der er også kun en enkelt processor i Amigaen; og processoren kan i virkeligheden ikke gøre flere ting af gangen.

Tricket består i, at når flere programmer skal køre på samme tid, bliver efter tur arbejdet en smule på hvert program. I eksemplet ovenfor bliver bolden først rykket et enkelt billedpunkt frem, så ses efter, om der er blevet indtastet et tegn i tekstbehandlingen, og det bliver i så fald udlæst, og så bliver uret rykket et sekund frem. Derpå begynder det hele forfra osv. osv. Problemet ved det er, at Amigaen bliver langsommere, når den skal klare mange delopgaver.

Hvert af disse jobs, som Amigaen skal klare "samtidigt", kaldes et task. Derfor kalder man den samtidige bearbejdning af flere tasks for "multitasking". Hvert task, der bliver tilføjet til processorens opgaverække, får tildelt en bestemt tid, hvori den bliver bearbejdet ved hvert gennemløb. Man kan også variere tiden, så tidskritiske programmer kan blive tilstrækkeligt bearbejdet.

*Hvordan processorens tidsinddeling fungerer, er ikke så vigtigt for programmøren. Man kan skrive og starte et program, uden at tage hensyn hertil. Det vil automatisk køre "ved siden af". Herunder skal man lægge mærke til, at programmet enten skal startes fra CLI'en med "run" eller fra workbench'en. Et program, der kun startes med navn, optager hele processorens tid, som det kan få fra CLI'en og vender, først efter at have afsluttet sit arbejde, tilbage til CLI'en. Hvis man starter det med "run", har man stadig direkte rådighed over CLI'en.*

Der findes endnu en begrænsning af muligheden for multitasking, der gælder specielt for assembler-programmer. Bortset fra anvendelsen af ekstra lagerplads, der ubetinget skal reserveres på forhånd, bør man ikke tilgå hardware-registrene direkte. I stedet skal de tilsvarende funktioner fra bibliotekerne anvendes. Grunden hertil er simpel:

Hvis man f.eks. har defineret printerporten som indlæsningssted, og indtaster data, kunne det jo ske, at et eller andet andet task mente, at det skulle printe. Ledningerne ville blive omstillet til udlæsning, og data blive udlæst. Derefter forsøger programmet igen at indlæse data, hvad der ikke længere vil lykkes.

Det er kun et simpelt eksempel, men det anskueliggør problemet. Der findes langt værre effekter, der kan optræde som følge af unkontrolleret flerdobbeltprogrammering af nogle hardware-registre. Hvis man alligevel lader et program tilgå harware-registrene direkte (hvad der kan indebære visse fordele), så er det bedre at sørge for, at det programmet kører alene.

# 2. Processoren MC68000

Den i Amigaen indbyggede processor, MC68000 er en meget effektiv komponent, der også anvendes i Atari ST og i Macintosh. Det er en 16/32 bits processor, dvs. den kan ganske vist arbejde med 32 bits, men den har "kun" en 16 bits bred databus og en 24 bits adressebus. Den kan derfor direkte tilgå et adresseområde på 2↑24=16777216 bytes, altså 16 Mbytes.

Processoren i Amiga kører med ca. 7,1 megahertz, og er således meget hurtig, hvad der er nødvendigt, for et så arbejdsintensivt job, som det i Amigaen. Den bliver i Amiga understøttet effektivt af nogle periferikomponenter, hvad der aflaster den betydeligt ved mange opgaver. Således bliver grafikudlæsning, tonefrembringelse og ind-/udlæsning delvis overtaget af disse komponenter, og giver således processoren tid til at bekymre sig om beregninger.

## 2.1. Registre

Udover det normale RAM-lager har processoren nogle interne registre. Der er 8 dataregistre (D0-D7), 8 adresseregistre (A0-A7), et statusregister (SR), to stackpointere (USP og SSP) og adressetælleren (ProgramCounter=PC).

Dataregistrene, adresseregistrene og adressetælleren er 32 bits brede, statusregistret 16 bits. Registrene tilgås ikke som en lagercelle, da de ikke ligger i lageret, men i processoren selv. Som De senere skal se under adresseringsmåder og i kommandoordforrådet, findes der specielle kommandoer til dem.

Dataregistrene anvendes til data af alle arter. Operationer med bytes (8 bits), ord (16 bits) og lange ord (32 bits) er mulige, så deres anvendelse er særdeles fleksibel.

Adresseregistrene bliver anvendt til lagring og forarbejdning af adresser. De kan insættes som pointere på tabeller o.a. Herunder er kun operationer med ord og lange ord mulige.

Adresseregister A7 har en særlig rolle. Registeret anvendes altid som stackregister (stackpointer=SP) og kan således ikke anbefales til almindelig anvendelse. Hvilken af de to mulige stacke, der peges på, afhænger af processorens øjeblikkelige funktionsmåde. Mere herom senere.

Stacken, på hvis aktuelle position SP peger, bliver anvendt til opbevaring af interne data, der kun skal lagres midlertidigt. Stacken fungerer som en stabel notitser på et skrivebord. Det der sidst blev lagt i bunken, tages først op. Den måde at stable på kalder man LIFO (Last In, First Out). Der findes også en anden type, FIFO- (First In, First Out) stacken, der imidlertid ikke anvendes af processoren selv.

Hvordan man håndterer disse registre og dermed SP, og hvordan man arbejder med stacken, kan De bl.a. finde i det næste kapitel. Men tilbage til registrene.
Statusregistret SR spiller en ganske særlig rolle for maskinsprogsprogrammering. I det ord indeholdes på 10 bits nogle vigtige informationer, der giver oplysninger om processorens status. Man underopdeler dette ord i den laveste byte, bruger-byten, og den højeste, system-byten. De bits, der får en betydning, kaldes flag. Dvs., at der angives en bestemt tilstand, når bitten er sat.
User-byten omfatter 5 flag, der har følgende betydning:

| Bit | Navn | Betydning |
|---|---|---|
| 0 | (C, carry) | Overførelsesbit. Bliver i givet fald ændret ved sammenknytninger, beregninger og forskydninger. |
| 1 | (V, overflow) | Ligner carry, men viser en fortegnsændring resp. en overførelse af bit 6 til bit 7. |
| 2 | (Z, zero) | Bliver sat, hvis resultatet af en operation er nul. |
| 3 | (N, negativ) | Sættes, hvis resultatet af en operation er negativt (højeste bit sat). |
| 4 | (X, extended) | Sættes i givet fald ved aritmetiske operationer ligesom carry. |
| 5-7 | | Ubenyttede |

I systembyten er ligeledes 5 bits af betydning:

| Bit | Navn | Betydning |
|---|---|---|
| 8 | I0 | Interrupt-maske. Hermed bliver interrupt- |
| 9 | I1 | niveauet indstillet fra 0 til 7, hvor 0 |
| 10 | I2 | er det laveste og 7 den højeste prioritet, hvor interrupts er tilladt. |
| 11 | | Ubenyttet. |
| 12 | | Ubenyttet. |
| 13 | (S, supervisor) | I denne bit indeholdes den øjeblikkelige funktionsmåde som processoren befinder sig i (0=user-, 1=supervisor-funktionsmåde). |
| 14 | | Ubenyttet. |
| 15 | (T, trace) | Er denne bit sat, så befinder processoren sig i enkelttrins-funktionsmåde. |

Her er en oversigt over hele statusordet:

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Bit:** | **15** | **14** | **13** | **12** | **11** | **10** | **9** | **8** | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |
| **Navn:** | **T** | **-** | **S** | **-** | **-** | **I2** | **I1** | **I0** | **-** | **-** | **-** | **X** | **N** | **Z** | **V** | **C** |

Her dukker begreber som funktionsmåde (modus) og interrupt op, som man ikke bør lade sig forvirre af. Disse ting bliver behandlet senere i et kapitel om processorens funktionstilstand.

## 2.2. Lager-adressering

Processoren råder i en normalt udbygget Amiga 500 eller 1000 over 512 Kbytes RAM, Amiga 2000 endda over 1 Mbyte, hvori den efter forgodtbefindende kan aflæsse data. Men hvordan kommer den igen i forbindelse med disse data? Vi vil se på denne problemstilling.

Hvis man programmerer i BASIC, behøver man ikke at gøre sig nogen forestilling om lagerets opdeling. Man skriver MAERKE%=1, og værdien er lagret.

Der er to måder at realisere processen på i assembler:

1) **man lægger værdien ind i et af data- eller adresseregistrene, eller**
2) **man skriver den ind i en lagercelle.**

For at vise begge måder på een gang, foregriber vi begivenhederne, og betragter den første maskinsprogsordre, der er den mest almindelige : MOVE. Med denne ordre bliver, som navnet siger, en værdi forskudt.Ordren tilføjes desuden to angivelser: kilde og mål.

Lad os betragte det ovenstående eksempel, når vi anvender MOVE-kommandoen:

**ved 1) MOVE #1,D0**

lægges værdien 1 i dataregister D0. Som det ses, angives først kilde- og så mål-operanden. Derfor er kommandoen MOVE D0,#1 ikke mulig.

**ved 2) MOVE #1,$1000**

lægges den bekendte værdi 1 ind i lagercelle $1000. Adressen er valgt vilkårligt. Nomalt dukker ingen adresser op i programmer , der er skrevet med assemblersprog, fordi man indsætter såkaldte labels, der henviser til et bestemt sted i programmet. Således skriver man, hvis man vil skrive 1 i en lagercelle i programmet:

```
...
MOVE #1,MAERKE

...
MAERKE: DC.W 1
```

Det drejer sig om to steder i programmet. Den øverste linie udfører den sædvanlige MOVE-kommando, hvorunder måladressen er "MAERKE". Den label dukker så igen op og markerer den lagercelle, hvori værdien skal skrives.

Kommandoen DC.W 1 er en såkaldt pseudo-op. Det betyder, at det ikke er en maskinordre til processoren, men en kommando til assembleren. DC står for "declare", frit oversat "reserver". Tilføjelsen .W betyder, at det i dette tilfælde drejer sig om et dataord (.W som "word"). Andre muligheder er .B for byte (8 bits) og .L for et langt ord (32 bits).

Tilføjelsen (.B, .W eller .L) kan anvendes ved de fleste maskinordrer. Udelader man den, så forudsætter assembleren en normal ordlængde og assemblerer den som .W. Hvis man vil anvende et langt ord, så skrives MOVE.L #$12345678,D0, en byte skrives med MOVE.B #12,D0. Det er noget, der *ubetinget* skal overholdes:

**Oplysning: Hvis lageret tilgås med ord- eller langord-operationer, så skal adressen altid være et lige tal (slutciffer 0,2,4,6,8,A,C,E)!**

Det kan man, ved anvendelse af en assembler, opnå med kommandoen "EVEN". Den er altid nødvendig, når man skriver noget i retning af det følgende

```
...
VAERDI1:  DC.B 1
VAERDI2:  DC.W 1
```

Hvis lagercellen VAERDI1 ligger på en lige adresse, må VAERDI2 nødvendigvis ligge på en ulige. Men hvis man skriver et EVEN, bliver om nødvendigt tilføjet et "fyldebyte" (0) af assembleren, og adressen bliver igen lige.

```
...
VAERDI1:  DC.B 1
EVEN
VAERDI2:  DC.W 1
```

Tilbage til eksempelet med adresseringsmåderne. De ovenfor anførte varianter, kan sammenlignes med BASIC-kommandoen MAERKE%=0, hvorunder %-tegnet henviser til en heltals-værdi.

Vi går et skridt videre og "oversætter" MAERKE%=VAERDI%. Jeg antager, at De allerede kender løsningen, ikke sandt? Så sammenlign Deres id med følgende:

```
          MOVE VAERDI, MAERKE
          ...
          ...
MAERKE:   DC.W 1
VAERDI:   DC.W 1
```

Her "moves" i stedet for den direkte værdi 1, kun indholdet af lagercellen på adressen "VAERDI", til adressen "MAERKE".

Vi har med små eksempler, lært fire adresseringsmetoder at kende, hvormed processoren kan tilgå lageret. Den første kendetegnes af et #-tegn og repræsenterer en direkte værdi. Adresseringsmåden hedder derfor "direkte adressering". Den er naturligvis, kun tilladt for kilde-operander!

Endnu en metode, hvor angives en direkte adresse (her "MAERKE" og "VAERDI") hedder simpelthen "absolut adressering". Den er mulig, både for kilde- og for mål-operander, som vi har set.

Den bliver underopdelt i to forskellige adresseringsmåder, som programmøren normalt ikke mærker noget til. Alt efter om den absolutte adresse er større eller mindre end $FFFF, altså kræver, eller ikke kræver et langt ord, kaldes den nemlig "absolut lang" ved adresser >$FFFF eller i modsatte tilfælde "absolut kort". Denne skelnen imellem, hvilken adresseringmåde, der er tale om, træffes normalt af assembleren, så De kun behøver at mærke Dem den absolute adressering generelt.

Den fjerde af de hidtil opdukkede adresseringsmetoder, hedder "dataregister direkte". Den blev præsenteret, som den første (MOVE #1,D0) i forbindelse med den direkte adressering og betyder, at et dataregister (her D0) bliver anvendt direkte.

Der findes flere end disse fire adresseringsmåder til 68000-processoren, nemlig hele 12. En måde hedder f.eks. "adresseregister direkte" og svarer til "dataregister direkte", blot bliver her anvendt et adresseregister. Således kan man med MOVE.L #MAERKE,A0 bringe mærkets adresse direkte ind i adresseregister A0.

Det var altså allerede fem adresseringsmåder, der alene vil give os noget at tage fat på. Men nu bliver det lidt mere kompliceret, men også mere interessant.

Lad os atter tage et eksempel fra programmeringssproget BASIC, nemlig følgende programstykke:

```
10 A=1000
20 POKE A,1
```

Variablen A bliver først tildelt den direkte værdi 1000. Det kan vi også gøre på assembler: MOVE.L #1000,A0. Her bliver den absolutte værdi A skrevet, som et langt ord i adresseregistret A0.

Nu til linie 20. Her bliver ikke tildelt en værdi til variablen A, men til den lagercelle, hvis adresse står i A. Det drejer sig altså om en indirekte tilgang, der let lader sig realisere i assembler:

```
MOVE.L #1000,A0    ;læg adressen i A
MOVE   #1,(A0)     ;og skriv 1 på denne adresse
```

Ved hjælp af parenteserne, der står omkring A0, angives adresseringsmetoden, som hedder "adresseregister indirekte". Adresseringsmetoden fungerer således kun med adresseregistre. Der findes ikke noget "dataregister indirekte".

Der eksisterer også nogle varianter af denne adresseringsmetode. Således kan man angive en afstandsværdi i assemblerkommandoen, der adderes til den adresse, som findes i adresseregistret, før der gås til den. MOVE #1,4(A0) indsat i ovenstående eksempel, vil altså skrive 1 i lagercelle 1000+4=1004. Denne afstandsværdi kan være indtil 16 bits bred, og bliver behandlet med fortegn. Derfor kan den have en værdi fra -32768 til +32767. Adresseringsmetoden hedder "adresseregister indirekte med 16-bits-afstandsværdi".

En udvidelse hertil er: "adresseregister indirekte med 8-bits-afstandsværdi", hvor afstandsværdien ikke er begrænset til 8 bits, men der dukker et register mere op. Det nye register regnes som en afstandsværdi, men er naturligvis også en variabel.

Lad os tydeliggøre det med et eksempel. Antag, at vi har en dataliste i programmet, opbygget på følgende måde:

```
       ...
LISTE:  DC.W 2          ;antal poster-1
        DC.W 1,2,3      ;liste-elementer.
```

Ved hjælp af MOVE.L #LISTE,A0 indlæser vi adressen på listen i adresseregister A0. Nu kan vi hente antallet af listeelementer i dataregistret med MOVE (A0),D0. For at kunne gå til det sidste element på listen, behøver vi en ordre. Det hele ser således ud:

```
CLR.L     D0              ;slet D0 helt
MOVE.L    #LISTE,A0       ;adresse på listen i A0
MOVE      (A0),D0         ;antal af elementer-1 i D0
MOVE      1(A0,D0),D1     ;og sidste element i D1

   ...
LISTE:    DC.W 2          ;antal poster-1
          DC.W 1,2,3      ;liste-elementer
```

Den sidste ordre griber altså til den byte, der ligger i 1+A0+D0, dvs. liste+1, hvor dataene begynder, plus indholdet af D0, i dette tilfælde 2.

Med den adresseringsmåde kan man øjensynlig gøre en hel del. Den er fremragende egnet til forarbejdning af lister og tabeller, som eksemplet viser. Har man ikke brug for nogen afstandsværdi, skal man indsætte et 0, hvad mange assemblere klarer af sig selv, hvis man f.eks. kun indtaster MOVE (A0,D0).

Af de to sidst præsenterede adresseringsmåder, findes en variant, der har en speciel egenskab. Her anvendes ikke et eller andet adresseregister, men programtælleren (PC). Denne måde anvendes først og fremmest, når man vil udforme sit program således, at det kan køre uden problemer i et hvilken som helst lagerområde. Således har (inden for 15-bits grænserne) begge de følgende kommandoer samme effekt:

**MOVE MAERKE,D0**

og

**MOVE MAERKE(PC),D0**

Denne skrivemåde er egentlig upræcis, da der i den øverste kommando med MAERKE, menes mærkets adresse, men i den nederste afstanden mellem kommandoen og mærket. Da det vil være for omstændeligt, hver gang at skulle beregne afstanden, accepterer en assembler det i denne form, og beregner selv den virkelige værdi.

Men lad os se på forskellen mellem den øverste og den nederste kommando. De har ganske vist samme effekt i programmet, men oversættes helt forskelligt af assembleren. Lad os antage, at programmet ligger fra adresse $1000 og mærket i adresse $1100. Den frembragte programkode er så den følgende:

```
$001000   30 39 00 00 11 00   MOVE MAERKE,D1
resp.
$001000   30 3A 00 FE         MOVE MAERKE(PC),D1
```

Den frembragte kode i den anden version, er to bytes kortere end i den første version. Yderligere opstår den effekt, at hvis De flytter koden til et lagerområde fra $2000 af, går den første version alligevel stadig til $1100, medens der med den PC-indirekte adressering, gås rigtigt til $2100: programmet er forskydeligt.

Det er den adresseringsmåde, som betegnes "programtæller indirekte med 16-bits afstandsværdi". Derudover findes, som omtalt, metoden "programtæller indirekte med 8-bits afstandsværdi", der tillader endnu et register som afstandsangivelse, også kaldet offset.

Har De talt med? Rigtigt, der mangler stadig to adresseringsmetoder. Også disse to bygger på indirekte adressering. Man har nemlig mulighed for, ved lagertilgang med "adresseregister indirekte", automatisk at forhøje eller formindske adresseregistret med 1.

Til automatisk forhøjelse bruges adresseringsmetoden "adresseregister indirekte med postinkrement", der forhøjer adresseregistret EFTER tilgangen, med det anvendte antal bytes. Hvis man altså skriver:

```
MOVE.L  #1000,A0
MOVE.B  #1,(A0)+
```

Så skrives ettallet på adresse 1000 og A0 forhøjes så med n. En sådan ordre er meget nyttig, når man vil udfylde et lagerområde med en bestemt værdi (f.eks. slette skærmbilledlageret). Til det brug kan man indlægge ordren i en løkke. - Hvordan det sker, ser vi senere.

Modstykket til postinkrement er adresseringsmåden "adresseregister indirekte med prædekrement". Her bliver det angivne adresseregister formindsket med n FØR tilgangen. Således bevirker

```
MOVE.L  #1000,A0
MOVE.B  #1,-(A0)
```

at der skrives 1 på adresse 999, fordi indholdet af A0 først blev formindsket, og derefter blev der skrevet 1.

Disse to adresseringsmetoder bliver anvendt til at forvalte stackpointeren (SP). Da stacken, fyldes ovenfra og nedefter, skriver man, for at lagre et ord, f.eks. fra D0 i stacken:

**MOVE.B D0,-(SP)**

og vil man tage det fra stacken, og igen lægge det i D0, skrives:

**MOVE.B (SP)+,D0**

Hermed peger stackpointeren altid på den byte, der sidst er lagt i stacken. Man må passe meget på, at ord- eller langord-tilgangen til stacken sker på en lige adresse. Når man vil lægge et ord i stacken, gøres det enten straks med ordbredde, eller man passer på, at det ikke resulterer i noget JSR eller BSR af byte'n. Ordrerne lægger nemlig den adresse, som de kommer fra, ind i stacken som et langt ord.

Med ovenstående skrivemåde sætter man adresseregister A7 i programkoden i stedet for SP, da det adresseregister altid anvendes som SP. Derfor kan man roligt skrive A7 i stedet for SP - programmet er det samme. Alligevel vil jeg anbefale at anvende SP, da det bliver mere overskueligt. Man opretter ofte egne stackområder i maskinprogrammer, som man så ligeledes kan tilgå på den måde. Så er det meget vigtigt, at kunne skelne tydeligt mellem processorstack og den personlige stack.

Det var de 12 adresseringsmetoder, som MC68000-processoren har. Her er de alle i resum:

| Nr. | Navn | Skrivemåde |
|---|---|---|
| 1 | dataregister direkte | Dn |
| 2 | adresseregister direkte | An |
| 3 | adresseregister indirekte | (An) |
| 4 | adresseregister indirekte med postinkrement | (An)+ |
| 5 | adresseregister indirekte med prædekrement | -(An) |
| 6 | adresseregister indirekte med 16-bits afstandsværdi | d16(An) |
| 7 | adresseregister indirekte med 8-bits afstandsværdi | d8(An,Rn) |
| 8 | absolut kort | xxxx.W |
| 9 | absolut lang | xxxxxxxx.L |
| 10 | umiddelbar | #"data" |
| 11 | programtæller indirekte med 16-bits afstandsværdi | d16(PC) |
| 12 | programtæller indirekte med 8-bits afstandsværdi | d8(PC,Rn) |

De her anvendte forkortelser betyder:

| | |
|---|---|
| **An** | adresseregister A0-A7 |
| **Dn** | dataregister D0-D7 |
| **d16** | 16-bits værdi |
| **d8** | 8-bits værdi |
| **Rn** | register D0-D7, A0-A7 |
| **"data"** | en værdi, der er indtil 32 bits bred, alt efter den anvendte databredde (.B,.W eller.L). |

Det var altså samtlige adresseringsmåder, som MC68000-processoren råder over. "Storebroderen" til denne processor, 32-bits processoren MC68020, har seks måder mere, som vi dog ikke skal gå ind på.

## 2.3. Funktionstilstande

I kapitlet om registre har vi lært statusregistret (SR) at kende. De enkelte bits i dette register angiver processorens funktionstilstand. Herunder skelnes mellem systembyte'n (bit 8-15) og userbyte'n (bit 0-7). Lad os begynde med systembyten og dens virkninger på processorens arbejde.

### 2.3.1. User- og supervisor-funktionsmåde

Er det ikke ejendommeligt, at processoren indplacerer en som user (bruger) eller supervisor (overvåger)? De to funktionstilstande er nemlig begge mulige, og det normale er user-funktionsmåde. Under denne funktionsmåde har man ikke tilladelse til at uddele forskellige maskinorder, og det i sin egen computer!

*Men ingen panik; denne begrænsning er ikke alene praktisk, men kan også omgås. Amiga-operativsystemet indeholder nemlig en funktion, hvormed processoren lader sig omstille til supervisor-funktionsmåde. Uden denne funktion er det meget omstændeligt, ja næsten umuligt, at foretage denne omstilling selv.*

Funktionsmåden bestemmes af bit 13 i statusregistret. Normalt er denne bit slettet (0), og processoren er i user-funktionsmåde. Nu kan der skrives direkte i statusregistret, men maskinordren er privilegeret, og kan derfor kun anvendes fra supervisor-funktionsmåden. Den ordre kan man derfor kun bruge til at omstille fra supervisor- til user-funktionsmåde, ved at slette supervisorbit'en med f.eks. AND #$DFFF,SR. Det kan ses, at sådan omstilling er det bedre at overlade til operativsystemet.

Hvad er nu forskellen på de to funktionsmåders anvendelse?

En første forskel er allerede blevet omtalt. Nogle maskinordrer, som f.eks. MOVE xx,SR, er privilegerede og kan derfor kun udføres i supervisor-funktionsmåden. Et forsøg på at gøre det i user-funktionsmåde, bliver straffet med afbrydelse af programmet gennem en såkaldt exception, en undtagelse. Disse exceptions skal vi se på senere. De byder på den eneste mulighed for at omstille til supervisor-funktionsmåde.

Endnu en forskel består i det stack-område, der anvendes. Ganske vist bliver A7 stadig anvendt som stackpointer, men der bliver benyttet et andet lagerområde. Sp ændres derfor altid ved omstilling fra den ene til den anden funktionsmåde. Man skelner mellem user-SP (USP) og supervisor-SP (SSP).

Også lagertilgange kan være funktionsmåde-afhængige. Processoren videregiver ved en sådan tilgang signaler til de perifere komponenter, der fortæller om processorens funktionsmåde. Herunder kan forskellige lagerområder i en 68000-computer blive privilegeret, så man ikke kan tilgå dem.

I supervisor-funktionsmåde kan det altså lade sig gøre, at udføre alle kommandoer og tilgå alle lagerområder. Af den grund kører operativsystemer mest i supervisor-funktionsmåde. Det opnår man ved anvendelse af exceptions.

## 2.3.2. Exceptions

Exceptions, på dansk undtagelser, optræder af forskellige grunde eller årsager. Hvis en sådan årsag optræder, så forgrenes der, alt efter begivenhedens art, til en bestemt rutine. Først bliver der dog foretaget nogle skridt:

1) **Statusregistret saves**
2) **S-bit'en i SR bliver sat (supervisor-funktionsmåde) og T bit'en slettes (intet Trace).**
3) **Programtælleren og user-SP saves**
4) **Exception-vektoren, der peger på den exception-rutine, som skal udføres, hentes.**
5) **Rutinen udføres.**

De omtalte vektorer, hvori der står adresserne på de forskellige rutiner til bearbejdelse af begivenheden, ligger helt i begyndelsen af lageret. Her er en oversigt over vektorerne og deres adresser:

| Nummer | Adresse | Anvendes ved |
|---|---|---|
| 0 | $000 | RESET: begyndelses-SSP |
| 1 | $004 | RESET: begyndelses-PC |
| 2 | $008 | bus error |
| 3 | $00C | address error |
| 4 | $010 | illegal ordre |
| 5 | $014 | division med nul |
| 6 | $018 | CHK-ordre |
| 7 | $01C | TRAPV-ordre |
| 8 | $020 | privilegiekrænkelse |
| 9 | $024 | trace |
| 10 | $028 | Axxx-ordreemulering |
| 11 | $02C | Fxxx-ordreemulering |
| | $030-$038 | reserveret |
| 15 | $03C | uinitialiseret interrupt |
| | $040-$05F | reserveret |
| 24 | $060 | uberettiget interrupt |
| 25-31 | $064-$083 | level 1-7 interrupt |
| 32-47 | $080-$0BF | TRAP-kommandoer |
| | $0C0-$0FF | reserveret |
| 64-255 | $100-$3FF | user-interrupt vektorer |

De enkelte begreber i ovenstående tabel, kræver vist lidt nærmere betragtning. Lad os gennemgå dem et for et.

**RESET: begyndelses-SSP**
Det er vel det letteste at forstå: ved reset bliver det lange ord, der står her, brugt som stackpointer for supervisor-funktionsmåden. På den måde kan man fastlægge stacken til RESET-rutinen.

**RESET: begyndelses-PC**
Ved et reset bliver den værdi, der står her, brugt som programtæller, dvs., RESET-rutinen startes på den adresse, der indeholdes heri.

**Bus error**
Denne exception udløses af hardwaren, resp. en coprocessor, hvis der f.eks. tilgås et reserveret, eller ikke eksisterende, lagerområde.

**Address error**
Fejlen optræder, når man tilgår en ulige lageradresse med et ord eller et langt ord.

**Illegal ordre**
Da alle MC68000's ordrer består af t ord, kan man således formulere 65536 "ordrer". Fordi processoren ikke forstår så mange ordrer, er nogle ord ugyldige eller illegale som ordrer. Hvis en sådan ordre optræder, udløses exceptionen.

**Division med nul**
Da processoren råder over en divisionsordre, og division med nul ikke er defineret matematisk, optræder denne exception, hvis der foretages en operation af den type.

**CHK-ordre**
Exceptionen optræder kun ved CHK-ordren. Ordren tester et dataregister for, om dets indhold er indenfor bestemte grænser. Hvis det ikke er tilfældet, udløses exception'en.

**TRAPV-ordre**
Udføres TRAPV-ordren, så udløses denne exception, hvis V-bit'en er sat i statusordet (bit 1).

**Privilegiekrænkelse**
Hvis en privilegeret ordre kaldes i user-funktionsmåde, så udløser det denne exception.

**Trace**
Hvis trace-bit'en (bit 15) er sat i statusordet, så bliver exceptionen udløst, efter hver udført ordre. På den måde kan man let realisere en enkelt-trins kørsel af maskinprogrammer.

**Axxx-ordre-emulering**
**Fxxx-ordre-emulering**
Disse to vektorer kan anvendes til et interessant trick. Hvis der nemlig kaldes en ordre, der begynder med $A resp. $F (f.eks. $A010 eller $F200, så hoppes til den rutine, der peges på ved hjælp af den tilsvarende vektor. I rutinerne kan der nu, alt efter resten af ordreordet, udløses funktioner, hvorved man på en måde kan skrive ordreudvidelser til processoren.

**Reserveret**
Disse vektorer anvendes ikke.

**Uinitialiseret interrupt**
Denne exception udløses, når et interrupt kommer fra en periferi-komponent, der ikke er blevet initialiseret.

**Uberettiget interrupt**
Et sådant optræder, når der forekommer en BUS-fejl under interrupt-bekræftelsen af den komponent, der skal udløses. I så tilfælde er det sandsynligvis kun om en forstyrrelse i det udløste interrupt.

**Level 1-7 interrupt**
Disse 7 vektorer peger på interrupt-rutiner af tilsvarende prioritet (level). Hvis det level, der indeholdes i statusordet, er højere end level'et i det interrupt, der optræder, så ignoreres det sidste.

**TRAP-ordrer**
Disse 16 vektorer anvendes, når der foreligger en tilsvarende TRAP-ordre. Det er derfor muligt med TRAP-ordrer fra TRAP #0 til TRAP #15.

**User-interrupt-vektorer**
Vektorerne anvendes til interrupts, der udløses af visse periferi-komponenter, som frembringer et selvstændigt vektornummer.

Vi vil ikke gå nærmere ind på exception-behandlingens hemmeligheder og muligheder på dette sted, for ikke at sprænge denne bogs rammer. Vi vil blot bemærke en ting til: exception-rutinerne afsluttes ved hjælp af ordren RTE (return from exception), hvorved den gamle tilstand bliver genoprettet, og user-programmet kører videre.

## 2.3.3. Interrupts

Interrupts udføres ligesom exceptions. Det er programafbrydelser, der udløses af hardware-komponenter, altså en periferi-komponent eller en ekstern omkobling.

I stausregistret ligger der i bits 8-10 en værdi mellem 0 og 7, der repræsenterer det såkaldte interrupt-level. Der findes nemlig 8 forskellige interrupt-muligheder, der hver har en bestemt prioritet. Hvis denne prioritet er højere end den værdi, der indeholdes i statusregistret, så udføres interruptet, eller det ignoreres.

Når der optræder et gyldigt interrupt, så forgrenes til den tilsvarende rutine, hvis adresse indeholdes i exception-vektor tabellen (se ovenfor).

Den type programafbrydelser er meget vigtige, når man vil synkronisere et program med det tilsluttede hardware. Således kan en kobling, der skal videregive data til computeren, signalere, at der foreligger en gyldig værdi,

ved hjælp af et interrupt. Interrupt-rutinen behøver blot at overtage den foreliggende værdi direkte. Metoden anvendes også ved betjening af serielle interfaces (RS232).

Vi skal senere gå ind på anvendelsen af interrupts. Her skal blot tillige nævnes, at en interrupt-rutine ligesom en exception-rutine afsluttes med RTE-ordren.

Før vi ser på statusregistrets user-byte, er her et par bemærkninger om trace-bit'en. Når denne bit er sat, bliver exceptions derefter, som før, udført normalt. Men i normal drift, uafhængigt af den aktuelle funktionsmåde, bliver der udløst en exception (nr. 9, trace) efter hver udført maskinordre, hvori så registrene osv. kan udnyttes. Metoden anvendes i debugger-programmer som K-SEKA.

## 2.3.4. Betingelsescodes

Når man skriver et program i et programmeringssprog, bliver det hele tiden nødvendigt med en betinget operation; således f.eks. i et BASIC-program:

**IF D1=2 THEN D2=0**

For at realisere det i et maskinprogram, skal man naturligvis først gennemføre sammenligningen:

**CMP #2,D1**

CMP betyder compare (engelsk: sammenligne) og bevirker en sammenligning af de to operander, her D1 med værdien 2. Men hvordan kan man udnytte værdien af denne sammenligning?

Til det formål findes de såkaldte betingelseskoder (CC=condition code), der forekommer i maskinsprogets forgreningsordrer. At forgrenes hedder i "computersproget" på engelsk "to branch". Derfor redder disse forgreningsordrer branch. Men det må naturligvis angives hvorhen og hvornår, der skal forgrenes.

Det enkleste tilfælde er den ubetingede forgrening. Den tilsvarende ordre hedder "BRA adresse", hvor BRA står for "branch always". Ordren er os naturligvis ikke til megen hjælp her.

Der er reserveret nogle bits i statusregistret til at opbevare resultatet af sammenligningsoperationen CMP indtil brugen. Lad os se på bit 2, zero-flaget (zero = nul). Denne bit sættes netop, når resultatet af den foregående operation var nul.

For at klarlægge sammenhængen mellem CMP og Z-flaget, må vi se på CMP-ordrens funktion. Kommandoen udfører nemlig i virkeligheden en subtraktion af kildeoperanden fra måloperanden. I eksemplet bliver værdien 2 trukket fra indholdet af dataregister D1. Resultatet bliver ganske vist smidt ud, men først sættes det tilsvarende flag.

Hvis register D1 i vort eksempel ligeledes indeholdt værdien 2, så er resultatet af subtraktionen altså nul. Det bevirker, at Z-flaget sættes, hvad der nu kan udnyttes af en passende forgreningsordre. Eksemplet ser således ud:

```
  ...
  CMP    #2,D1     ;sammenligning resp. subtraktion
  BNE    ULIG      ;forgrening, når ulig (Z slettet)
  MOVE   #0,D2     ;ellers udføres D2=0
ULIG:
  ...              ;hertil forgrenes der i givet fald.
```

NE står for "branch if not equal", der betyder så meget som "forgren, hvis ikke lig". Det er en klar funktion efter en CMP-ordre. I virkeligheden forgrener BNE, hvis Z-flaget er slettet (=0). Modstykket hertil er BEQ-ordren, branch if equal (Z=1).

Her er en liste over alle betingelseskoder, som muliggør en betinget forgrening med Bcc (cc = condition code):

| cc | Betingelse | Bits |
|---|---|---|
| T | true, altid, svarer til BRA | |
| F | false, aldrig, forgrener aldrig | |
| HI | higher, højere end | C' * Z' |
| LS | lower or same, lavere eller lig med | C + Z |
| CC, HS | carry clear, higher or same | C' |
| CS, LO | carry set, lower | C |
| NE | not equal, ulig | Z' |
| EQ | egual, lig | Z |
| VC | overflow clear, intet overløb | V' |
| VS | overflow set, overløb | V |
| PL | plus, positiv | N' |
| MI | minus, negativ | N |
| GE | greater or equal, større end eller lig | N*V + N'*V' |
| LT | less than, mindre end | N*V' + N'*V |
| GT | greater than, større end | N*V*Z'+N'*V'*Z' |
| LE | less or equal, mindre eller lig | Z + N*V'+ N'*V |

*** = logisk OG,   + = logisk ELLER,   ' = logisk IKKE**

Her er nogle eksempler på, hvordan dette store antal betingelser kan anvendes:

```
CMP #2,D1
BLS MINDRE_LIG
```

Der forgrenes, hvis indholdet af D1 <= 2 (D1 = 0, 1 eller 2). Den lignende ordre BLE derimod, forgrener i dette tilfælde også, hvis D1 er negativ. Det kan man se på, at der ved vurderingen af betingelsen anvendes V-bit'en (se ovenfor), der reagerer på fortegnsskift under operationen og sammenlignes med N-bit'en. Hvis begge bits er slettet, efter den med CMP gennemførte subtraktion (D1-2), så er resultatet forblevet positivt (N-bit=0, V-bit=0): betingelsen er ikke opfyldt.

Betingelserne EQ og NE er også vigtige til andre anvendelser. Således kan man fastslå, om bestemte bits i et dataord er sat, ved at skrive følgende sekvens:

```
...
AND #%00001111,D1      ;afmaske bits
BEQ INGEN              ;forgrener, hvis ingen af de fire
                       laveste bits er sat
CMP #%00001111,D1
BEQ ALLE               ;forgrener, når alle fire bits er sat
```

AND-ordren bevirker, at alle bits fra D1 sammenlignes med parameterens bits (her #%00001111). Hvis der er sat de samme bits i begge bytes, så bliver de overtaget af resultatet. Hvis blot en af de to bits er slettet, så bliver det også nul i resultatet. Deraf resulterer, at der i D1, efter AND-sammenknytningen, kun er sat de bits, der før var sat i de fire laveste bits.

*Det kaldes at afmaske. I ovenstående eksempel blev de underste fire bits afmasket, dvs., i resultatet vil kun disse fire bits fremkomme i den tilstand, som de var i. Alle andre bits bliver slettet ved AND-sammenknytningen. Selvfølgelig kan man her anvende vilkårlige bitkombinationer.*

Hvis der ikke længere er nogen bit sat i resultatet, så bliver det signaleret af zero-flaget, hvormed BEQ-ordren er opfyldt, og forgrener sig. I modsat fald fortsættes med den næste ordre, hvori D1 sammenlignes med %00001111. Hvis begge er ens, så var oprindelig mindst alle de fire underste bits sat, og den følgende BEQ-ordre forgrener sig.

Betingelserne CC og CS kan man, foruden ved CMP, også anvende til at fastslå, hvornår en HI-bit bliver skubbet ud af dataordet, under rotation af data, ved hjælp af ROL- eller ROR-ordrer.

Før vi går til MC68000's ordreordforråd, vil jeg give Dem et tip:

Med K-SEKA assembleren er man, på den enkleste måde, i stand til at afprøve alle mulighederne i hver enkelt ordre. Lad os som eksempel tage den

netop omtalte CMP-ordre. For at teste ordren med forskellige værdier, og få resultaterne direkte ved hjælp af flag'ene, skal man load'e SEKA-assembleren.

Efter angivelsen af workspace, f.eks. 100, trykker man på escape-tasten, og kommer således frem til editoren. Nu indtaster man (linienumrene sætter assembleren selv):

```
1 run:
2     cmp #10,d1
3     bra  run
```

Tryk escape igen, og lad dette program assemble med "a" og to gange return. Derefter sætter man med

```
>xpc <return>
$000000 run <return>
```

programtælleren på adressen "run". Nu kan man indstille dataregistret D1 med "xd1" og den ønskede værdi, og så lade CMP-ordren udføre ved hjælp af "s". Man kan så direkte aflæse indholdet af registret (D1 har naturligvis ikke ændret sig i dette eksempel) og også af flagene i "klartekst" (foruden angivelse af statusordet SR). For derefter at afprøve en anden værdi af D1, kan man indstille den igen med "xd1" og med "s2" BRA-ordren, der ikke virker på noget register, og atter lade CMP-ordren køre.

*I øvrigt bør man ikke starte dette program med "g run", for så kører det evigt eller lige til et reset, hvad der ikke ligefrem er ønskværdigt.*

Metoden er naturligvis ikke begrænset til CMP-ordren. På samme måde kan man afprøve enhver anden ordre. Hvis man skulle komme til at lave en fejl herunder, så viser SEKA det, idet den opfanger alle exceptions, og skriver dem ud i klartekst (f.eks. address error at MOVE #1,1001). Det kan også anbefales at gøre dette med de fleste ordrer, fordi man på den måde væsentlig lettere kommer til at forstå de enkelte ordrers funktion, end ved kun at læse næste kapitel.

## 2.4. Ordresættet

Nu er det vel ved at være på tide at beskrive MC68000's ordrer. Desværre er her ikke plads til en udtømmende beskrivelse af hver enkelt ordre. Derfor må jeg henvise til den specielle litteratur om MC68000, f.eks. "Processorbogen om 68000" fra DATA-BECKER. I bogen kan De også lære om den hardwaremæssige opbygning af processoren og dens periferi.

Sammen med ordrerne i de følgende tabeller er der, hvert sted anført de nødvendige parametre eller argumenter. Argumenternes skrivemåde betyder følgende:

| | |
|---|---|
| **Label** | en label resp. adresse |
| **Reg** | register |
| **An** | adresseregister n |
| **Dn** | dataregister n |
| **Kilde** | kildeoperand |
| **Maal** | måloperand |
| **<ea>** | adresse eller register |
| **#n** | direkte værdi |

Her er en liste over MC68000's programstyringsordrer:

| **Mnemonic** | | **Betydning** |
|---|---|---|
| **Bcc** | **label** | **Forgren betinget, alt efter betingelse.** |
| **BRA** | **label** | **Forgren ubetinget (ligesom JMP).** |
| **BSR** | **label** | **Forgren til et underprogram. Ligesom ved JSR bliver returadressen herunder lagret på stacken og der returneres med RTS.** |
| **CHK** | **<ea>,Dx** | **Afprøv et dataregister for dets græsner, udløs evt. CHK-instruction-exception'en** |
| **DBcc** | **reg,label** | **Afprøv betingelse, dekrementer og forgren. Denne ordre anvendes hyppigt til løkker.** |
| **JMP** | **label** | **Hop til adresse (ligesom BRA).** |
| **JSR** | **label** | **Hop til et underprogram. Også her lagres returhopadressen på stacken, RTS fører tilbage.** |
| **NOP** | | **Ingen funktion (No operation).** |
| **RESET** | | **Reset periferien (pas på!).** |
| **RTE** | | **Returnering fra en exception.** |
| **RTR** | | **Returnering med indlæsning af flag.** |
| **RTS** | | **Returnering fra et underprogram (efter BSR eller JSR).** |
| **Scc** | **<ea>** | **Sæt en byte til -1, hvis betingelsen er opfyldt.** |
| **STOP** | | **Stands udførelsen (pas på!).** |
| **TRAP** | **#n** | **Hop til en exception.** |
| **TRAPV** | | **Afprøv, om overløbsflaget er sat, så evt TRAPV-exception.** |

Nogle bemærkninger hertil:

Ved hop (JSR) resp. forgrening (BSR) til en underrutine, bliver den returhopadresse, hvortil der skal returneres, efter afslutning af underrutinen, lagret på stacken. Med RTS hentes adressen fra stacken, og der hoppes dertil.

Vi vil eksperimentere lidt. Skriv følgende "program" i SEKA:

```
run:
        pea     underprogram    ;adresse på stacken
        jsr     underprogram    ;underprogram-kald
        move.l (sp)+,d1         ;langt ord fra stacken til D1
        illegal                 ;afbrydelse i SEKA

underprogram:
        move.l (sp),d0          ;returhopadresse til D0
        rts                     ;og tilbage
```

Lad os se nærmere på dette program. Først lægges underprogrammets adresse på stacken med ordren PEA. Derefter kaldes underprogrammet med JSR-ordren. Returhopadressen, altså den adresse, hvor hoveprogrammet skal køre videre efter udførelse af underprogrammet, lægges i stacken.

I underprogrammet bliver det lange ord, som stackpointeren peger på, læst ind i dataregister D0. Derefter hentes returadressen igen fra stacken med RTS-ordren, og der hoppes dertil.

Efter returneringen til hovedprogrammet bliver det lange ord, der ligger øverst på stacken, hentet ned og bragt til D1. Derefter afbryder SEKA, hvis programmet blev kaldt af den, kørslen ved hjælp af ILLEGAL-ordren, og viser indholdet af registrene.

Prøv nu at assemble dette program med "a" og to gange return, og start det med "g run" og to gange return. SEKA melder sig straks igen med "illegal instruction" og angivelse af registerindholdet. Her kan man så se, at på D0 ligger den adresse, hvor der skal arbejdes videre efter RTS-ordren. Desuden indeholder D1 adressen på underprogrammet, hvad man kan afprøve med "? underprogram" og sammenligning af resultatet med indholdet af D1. For også at kunne sammenligne direkte, skal De indtaste "n run", hvorefter De får programmet disassembleret.

*Endnu en interessant information, som SEKA-assemblerens statuslinie indeholder, er stackpointerværdien. Som De ved, indeholder adresseregister A7 denne pointer. Men hvorfor udlæser SEKA denne værdi en gang til, når man kan aflæse den i adresseregister A7?*

Som De ser, bliver der udlæst to stackpointerværdier: USP og SSP. Da vi for øjeblikket stadig befinder os i user-funktionsmåden, er værdierne i A7 og USP identiske. Nu vil vi afprøve den anden værdi.

Til den ende indtaster De "x sr" og får som opfordring til indtastning, også kaldet prompt, indholdet af statusregistret udlæst. Vi indtaster nu "$2000". Når man atter indtaste "x", ser man, at ved siden af udlæsningen af statusregistret (SR), står nu bogstavet "s". Det betyder, at computeren befinder sig i supervisor-funktionsmåde.

Start programmet en gang til, og se på indholdet af D1, A7 og USP. A7 og D1 er som før, men A7 afviger fra USP-værdien. Den aktuelle stackpointer er nemlig ikke mere USP, men SSP, fordi vi befinder os i supervisor-funktionsmåde. Det kan man ikke blot se på det lille "s" i statuslinien, men også på, at A7 og SSP viser den samme værdi.

Tilbage til betragtningen af de ovenfor anførte styringsordrer for MC68000.

STOP og RESET er ordrer med så dybtgående virkning, at de kun kan anvendes privilegeret (fra supervisor-funktionsmåde). Også selv om man kobler om til supervisor-funktionsmåde med SEKA, bør man ikke afprøve disse order, hvis man endnu har vigtige data i computeren, som endnu ikke er lagret. (Stol på guruen!)

TRAP-ordren får yderligere et nummer mellem 0 og $F, som bestemmer den TRAP-vektor, der skal anvendes (adresserne $0080-$00BF), og dermed den exceptions-rutine, der skal kaldes. Visse operativsystemer på 68000-basis anvender ordren til at kalde operativsystemfunktioner. Vi skal senere beskæftige os med denne ordre.

I det lille programeksempel, hvor vi sammenlignede to tal, blev med CMP gennemført en aritmetisk operation, nemlig subtraktion, som naturligvis også er mulig med et rigtigt resultat (SUB-ordre). Over for det står muligheden af addition (ADD-ordren). Med disse to operationer er mulighederne udtømt i en almindelig 8-bits processor som 6502"eren. Men MC68000 kan meget mere. Den kan tillige multiplicere, dividere, og gennemføre alle disse operationer med forskellige datastørrelser.

Til de fleste af operationerne kræves to parametre, således f.eks. ved ADD-ordren. Skrivemåden lyder:

**Add kilde, mål**

hvorunder kilde og mål kan være register- eller lageradresser. Kilde desuden også en direkte værdi (#n). Resultatet af operationen findes i målregistret, resp. mållageradressen. Det gælder for alle operationer af denne type. Også disse ordrer kan afprøves direkte ved hjælp af SEKA-assembleren, hvorunder anvendelsen af et register som mål, er anbefalelsesværdigt.

Her en oversigt over de aritmetiske operationer med heltal:

| Mnemonic | | Betydning |
|---|---|---|
| ADD | kilde,maal | Binær addition. |
| ADDA | kilde,An | Binær addition til et adresseregister. |
| ADDI | #n,<ea> | Addition med en konstant. |
| ADDQ | #n,<ea> | Hurtig addition af en konstant, der kun må være fra 1 til 8. |
| ADDX | kilde,maal | Addition med overførelse til X-flag. |
| CLR | <ea> | Sletning af en operand. |
| CMP | kilde,maal | Sammenligning af to operander. |
| CMPA | <ea>,An | Sammenligning med et adresseregister. |
| CMPI | #n,<ea> | Sammenligning med en konstant. |
| CMPM | kilde, maal | Sammenligning af to lageroperander. |
| DIVS | kilde, maal | Fortegnsrigtig division af 32-bit måloperanden med 16-bit kildeoperanden. Resultatet af divisionen ligger derefter i målets LO-ord, resten efter divisionen i HI-ordet. |
| DIVU | kilde, maal | Division uden fortegn, ligesom DIVS. |
| EXT | Dn | Fortegnsrigtig udvidelse til dobbelt længde. |
| MULS | kilde,maal | Fortegnsrigtig multiplikation af to ord til et langt ord. |
| MULU | kilde, maal | Multiplikation uden fortegn ligesom MULS. |
| NEG | <ea> | Negation af en operand (to-komplement). |
| NEGX | <ea> | Negation af en operand med overførelse. |
| SUB | kilde,maal | Binær subtraktion. |
| SUBA | <ea>,An | Binær subtraktion fra et adresseregister. |
| SUBI | #n,<ea> | Subtraktion af en konstant. |
| SUBQ | #n,<ea> | Hurtig subtraktion af en 3-bit konstant. |
| SUBX | kilde, maal | Subtraktion med overførelse til X-flaget. |
| TST | <ea> | Testning af en operand og i givet fald sætning af N- og Z-flag. |

I tilgift til bearbejdningen af heltal kan processoren også operere direkte med de såkaldte BCD-tal. BCD står for Binary Coded Decimal, og betyder, at der arbejdes med decimaltal. Med den metode bliver der kun lagret cifre mellem 0 og 9 i hver halvbyte, så man bedre kan bearbejde disse tal. Hertil findes følgende ordrer:

| Mnemonic | | Betydning |
|---|---|---|
| ABCD | kilde, maal | Addition af to BCD-tal. |
| NBCD | kilde, maal | Negation af et BCD-tal (nikomplement). |
| SBCD | kilde, maal | Subtraktion af to BCD-tal. |

Jeg anbefaler, at man også afprøver disse ordrer, da håndteringen af BCD-tal ganske vist er relativt enkel, men uvant. Man skal derfor passe på, at man ved indtastninger med "x" også virkelig indtaster BCD-tal, da resultaterne af operationerne ellers er ukorrekte. Man kan lettest sikre sig det, hvis man indlæser tallene decimalt, men med foranstillet "$"-tegn.

Dernæst kommer de logiske operationer, som vi til dels kender fra BASIC. Med disse operationer kan to værdier sammenknyttes binært, altså bit for bit.

| Mnemonic | | Betydning |
|---|---|---|
| AND | kilde, maal | Logisk OG. |
| ANDI | #n,<ea> | Logisk OG med en konstant. |
| EOR | kilde, maal | Eksklusiv ELLER. |
| EORI | #n,<ea> | Eksklusiv ELLER med en konstant. |
| NOT | <ea> | Inversion af en operand. |
| OR | kilde, maal | Logisk ELLER. |
| ORI | #n,<ea> | Logisk ELLER med en konstant. |
| TAS | <ea> | Afprøv en byte og sæt bit 7. |

Det resultat, der kommer ud af en sådan sammenknytning, kan man også erfare direkte med SEKA, idet man f.eks. indtaster

**?$55 & $AF**

**og får $05.**

Også enkelte bits kan manipuleres direkte. Dertil er følgende ordrer til rådighed:

| Mnemonic | | Betydning |
|---|---|---|
| **BCHG** | **#n,<ea>** | **Forandrer bit n (0 bliver til 1 og omvendt).** |
| **BCLR** | **#n,<ea>** | **Slet bit n.** |
| **BSET** | **#n,<ea>** | **Sæt bit n.** |
| **BTST** | **#n,<ea>** | **Test bit n, resultatet står i Z-flaget** |

Ordrerne er, frem for alt, af betydning ved håndtering og udnyttelse af data, der kommer fra en periferikomponent. Ved disse data spiller de enkelte bits ofte en særlig rolle. Det skal vi lære mere om i et senere kapitel.

Endvidere kan processoren forskyde og rotere en operand inden i sig, (n betyder et register eller en direkte værdi med #, der angiver antallet af forskydninger):

| Mnemonic | | Betydning |
|---|---|---|
| **AS** | **n,<ea>** | **Aritmetisk forskydning til venstre (*2↑n).** |
| **ASR** | **n,<ea>** | **Aritmetisk forskydning til højre (/2↑n).** |
| **LSL** | **n,<ea>** | **Logisk forskydning til venstre.** |
| **LSR** | **n,<ea>** | **Logisk forskydning til højre.** |
| **ROL** | **n,<ea>** | **Rotation til venstre.** |
| **ROR** | **n,<ea>** | **Rotation til højre.** |
| **ROXL** | **n,<ea>** | **Rotation til venstre med overførelse til X-flaget.** |
| **ROXR** | **n,<ea>** | **Rotation til højre med overførelse til X-flaget** |

Alle disse ordrer tjener til at forskyde en byte, et ord eller et langt ord, binært til venstre eller til højre. Hvorfor det er ensbetydende med en multiplikation resp. divsion med en potens af to, kan man se af den binære repræsentation af et eksempel.

Vi tager en byte med indholdet 16. I binær repræsentation er det:

**%00010000 = 16**

Nu forskyder vi byten en gang til venstre, hvorunder vi indsætter et nul til højre. Resultatet:

**%00010000 forskudt mod venstre giver**
**%00100000 = 32, altså 16*2**

Flerdobbelt forskydning giver således også flerdobbelt fordobling, hvorunder der ved n pladsers forskydning bliver multipliceret med 2↑n.

Tilsvarende gælder forskydning til højre. Her skal man lægge mærke til en ejendommelighed. Som eksempel en byte med indholdet 5:

**%00000101 = 5, n plads forskydning til højre giver**
**%00000010 = 2**

Som De ser, kommer der ikke 2,5 ud af det. Hvordan skulle det også vises? Resultatet af en sådan division giver altså altid kun en heltalsværdi, eventuelle cifre efter kommaet falder væk. Hvis man anvender DIV-ordren, så får man, ved siden af resultatet, også divisionsresten, så der ikke går noget tabt. Fordelen ved forskydningsmetoden frem for DIV-ordren er dog, at den for det første er hurtigere, og for det andet kan få hele lange ord som resultater.

Efter at forskydningens principper er afklaret, er der nu kun det spørgsmål tilbage, hvorfor det er nødvendigt med mere end to ordrer. Det ligger i, at der er flere typer af forskydninger.

For det første skelner man mellem at forskyde, på engelsk "to shift", og at rotere. Forskellen er blot, at ved forskydninger bliver der altid lagt et nul ind på den bit, der bliver fri. Ved rotering derimod, indsættes en bestemt værdi. Således bliver ved ordren ROR resp ROL den bit, der bliver skubbet ud forneden resp. foroven, atter skubbet ind på den modsatte side. Ved ROXR- resp ROXL-ordren tager denne bit omvejen over X-flaget. Indholdet af flaget bliver altså skubbet ind i den roterede værdi, og den "udskubbede" værdi bliver lagret i flaget.

Også ved forskydninger er der to varianter: aritmetisk og logisk forskydning. Den logiske forskydning har vi allerede lært at kende: der bliver altid skubbet et nul ind, og den bit, der sidst blev skubbet ud, står både i C- og i X-flaget.

Ved den aritmetiske forskydning bliver den mest betydende bit, der jo repræsenterer fortegnet, ganske vist skubbet med, men ved ASR bibeholdes den. Det har den fordel, at fortegnet bevares ved anvendelse af disse ordrer til division, så at -10/2 også giver -5. Hvis der alligevel optræder et fortegnsskift p.g.a. over- eller underløb, så bliver det meldt af det tilsvarende flag, V-flaget, der bliver sat ved et fortegnsskift. *Ved logisk forskydning bliver dette flag altid slettet.*

Og nu kommer vi til de ordrer, der bevæger dataene i computeren. Det er egentlig de vigtigeste ordrer i enhver processor, for hvordan skulle man ellers kunne bearbejde data?

| Mnemonic | | Betydning |
|---|---|---|
| EXG | Rn,Rn | Ombytning af to registerindhold (må ikke forveksles med SWAP). |
| LEA | <ea>,An | Indladning af en effektiv adresse i adresseregister An. |
| LINK | An,#n | Opbyg et stackområde. |
| MOVE | kilde, maal | Overfør en værdi fra kilde til maal. |
| MOVE | SR,<ea> | Overfør statusregisterindholdet. |
| MOVE | <ea>,SR | Overfør statusregisterindholdet. |
| MOVE | <ea>,CCR | Load flag. |
| MOVE | USP,<ea> | Overfør user-stackpointeren. |
| MOVE | <ea>,USP | Overfør user-stackpointeren. |
| MOVEA | <ea>,An | Overfør en værdi til adresseregister An. |
| MOVEM | Regs,<ea> | Overfør flere registre på en gang. |
| MOVEM | <ea>,Regs | Overfør flere registre på en gang. |
| MOVEP | kilde,maal | Overfør data til periferien. |
| MOVEQ | #n,Dn | Overfør hurtigt en 8-bits konstant til dataregister Dn. |
| PEA | <ea> | Læg en adresse i stacken. |
| SWAP | Dn | Ombyt registerhalvdelene (øverste 16 bits med de nederste). |
| UNLK | An | Opbyg et stackområde. |

Kommandoerne LEA og PEA er meget yndede til at indlæse adresser i et adresseregister, resp. til at lægge dem i stacken. Kommandoen

```
LEA  label,A0
```

indlæser adressen på label'en "label" i adresseregister A0. Det svarer i sin funktion nøje til ordren

```
MOVE.L  #label,A0
```

Tilsvarende betyder

```
PEA  label
```

bare at adressen på label'en lægges i stacken. Det erstatter på samme måde ordren

**MOVE.L #label,-(SP)**

Særlig interessant bliver LEA-ordren, når man ikke blot angiver en label, som effektiv adresse, men f.eks. en adresse, der er frembragt ved indirekte adressering. Et eksempel:

**LEA 1(A0,D0),A1**

Den adresse, der bliver frembragt ved den ovenstående adressering gennem addition af 1 (direktværdi-offset)+A0+D0, ligger så i A1. Eksemplet kan ikke frembringes med en MOVE-ordre. En tilsvarende programdel vil være:

```
MOVE.L   A0,A1
ADD.L    D0,A1
ADDQ.L   #1,A1
```

Som De kan se, byder LEA-ordren på interessante muligheder.

Det var alle MC68000's ordrer. Ved kombination med diverse adresseringsmetoder kan et stort antal ordrer sammenstilles, for at gøre et program så effektivt som muligt.

Følgende tabel, viser endnu engang alle MC68000'erens ordrer med de mulige adresseringsmåder og indflydelsen på flagene. De anvendte symboler betyder:

| | | |
|---|---|---|
| **x=tilladt** | **q=kun som kilde** | **z=kun som mål** |
| **-=upåvirket** | **0=slettet** | ***=modificeret tilsvarende** |
| **l=sat** | **u=ubestemt** | **p=privilegeret** |

| Mnemonic | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | XNZVC | P |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ABCD | x | | | | x | | | | | | | | | |
| ADD | q | q | x | x | x | x | x | x | x | q | q | q | ***** | |
| ADDA | x | x | x | x | x | x | x | x | x | x | x | x | ----- | |
| ADDI | x | | x | x | x | x | x | x | x | | | | ***** | |
| ADDQ | x | x | x | x | x | x | x | x | x | | | | ***** | |
| ADDX | x | | | | x | | | | | | | | ***** | |
| AND | q | | x | x | x | x | x | x | x | q | q | q | -**00 | |
| ANDI | x | | x | x | x | x | x | x | x | | | | -**00 | |
| ASL, ASR | x | | x | x | x | x | x | x | x | | | | ***** | |

| Mnemonic | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | XNZVC | P |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Bcc | | | | | | | | | | | | | ----- | |
| BCHG | x | | x | x | x | x | x | x | x | | | | --*-- | |
| BCLR | x | | x | x | x | x | x | x | x | | | | --*-- | |
| BRA | | | | | | | | | | | | | ----- | |
| BSET | x | | x | x | x | x | x | x | x | | | | --*-- | |
| BSR | | | | | | | | | | | | | ----- | |
| BTST | x | | x | x | x | x | x | x | x | z | x | x | --*-- | |
| CHK | x | | x | x | x | x | x | x | x | x | x | x | -*uuu | |
| CLR | x | | x | x | x | x | x | x | x | | | | -0100 | |
| CMP | x | x | x | x | x | x | x | x | x | x | x | x | -**** | |
| CMPA | x | x | x | x | x | x | x | x | x | x | x | x | -**** | |
| CMPI | x | | x | x | x | x | x | x | x | | | | -**** | |
| CMPM | | | x | | | x | x | x | x | | x | x | -*** | |
| cpGEN | | | | | | | | | | | | | ----- | |
| DBcc | | | | | | | | | | | | | ----- | |
| DIVS | x | | x | x | x | x | x | x | x | x | x | x | -***0 | |
| DIVU | x | | x | x | x | x | x | x | x | x | x | x | -***0 | |
| EOR | x | | x | x | x | x | x | x | x | | | | -**00 | |
| EORI | x | | x | x | x | x | x | x | x | | | | -**00 | |
| EORI CCR | | | | | | | | | | | | | ***** | |
| EORI SR | | | | | | | | | | | | | ***** | |
| EXG | | | | | | | | | | | | | ----- | |
| EXT | | | | | | | | | | | | | -**00 | |
| EXTB | | | | | | | | | | | | | -**00 | |
| ILLEGAL | | | | | | | | | | | | | ----- | |
| JMP | | | x | | | x | x | x | x | | x | x | ----- | |
| JSR | | | x | | | x | x | x | x | | x | x | ----- | |
| LEA | | | x | | | x | x | x | x | | x | x | ----- | |
| LINK | | x | | | | | | | | | | | ----- | |
| LSL, LSR | | | x | x | x | x | x | x | x | | | | ***0* | |
| MOVE | x | q | x | x | x | x | x | x | x | q | q | q | -**00 | |
| MOVEA | x | x | x | x | x | x | x | x | x | x | x | x | ----- | |
| MOVE to CCR | x | | x | x | x | x | x | x | x | x | x | x | ***** | |
| MOVE from SR | x | | x | x | x | x | x | x | x | | | | ----- | P |
| MOVE to SR | x | | x | x | x | x | x | x | x | x | x | x | ***** | P |
| MOVE USP | | x | | | | | | | | | | | ----- | P |

| Mnemonic | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | XNZVC | P |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| MOVEM | | | x | q | z | x | x | x | x | | q | q | ----- | |
| MOVEP | q | z | | | | | | | | | | | ----- | |
| MOVEQ | z | | | | | | | | | | | | -**00 | |
| MULS | x | | x | x | x | x | x | x | x | x | x | x | -**00 | |
| MULU | x | | x | x | x | x | x | x | x | x | x | x | -**00 | |
| NBCD | x | | x | x | x | x | x | x | x | | | | *u*u* | |
| NEG | x | | x | x | x | x | x | x | x | | | | ***** | |
| NEGX | x | | x | x | x | x | x | x | x | | | | ***** | |
| NOP | | | | | | | | | | | | | ----- | |
| NOT | x | | x | x | x | x | x | x | x | | | | -**00 | |
| OR | q | | x | x | x | x | x | x | x | q | q | q | -**00 | |
| ORI | x | | x | x | x | x | x | x | x | | | | -**00 | |
| PEA | | | x | | | x | x | x | x | | x | x | ----- | |
| RESET | | | | | | | | | | | | | ----- | P |
| ROL, ROR | | | x | x | x | x | x | x | x | | | | -**0* | |
| ROXL, ROXR | | | x | x | x | x | x | x | x | | | | -**0* | |
| RTE | | | | | | | | | | | | | ----- | P |
| RTR | | | | | | | | | | | | | ***** | |
| RTS | | | | | | | | | | | | | ----- | |
| SBCD | x | | | | x | | | | | | | | *u*u* | |
| Scc | x | | x | x | x | x | x | x | x | | | | ----- | |
| STOP | | | | | | | | | | x | | | ----- | |
| SUB | q | q | x | x | x | x | x | x | x | q | q | q | ***** | |
| SUBA | x | x | x | x | x | x | x | x | x | x | x | x | ----- | |
| SUBI | x | | x | x | x | x | x | x | x | | | | ***** | |
| SUBQ | x | x | x | x | x | x | x | x | x | | | | ***** | |
| SUBX | x | | | | x | | | | | | | | ***** | |
| SWAP | x | | | | | | | | | | | | -**00 | |
| TAS | x | | x | x | x | x | x | x | x | | | | -**00 | |
| TRAP | | | | | | | | | | x | | | ----- | |
| TRAPV | | | | | | | | | | | | | ----- | |
| TST | x | | x | x | x | x | x | x | x | | | | -**00 | |
| UNLK | | x | | | | | | | | | | | ----- | |

# 3. Hvordan man arbejder med assemblere

Alle ordrer til MC68000-processoren, som vi har lært at kende, kan den slet ikke forstå i den givne form. F.eks. siger bogstaverne "MOVE" overhovedet ikke processoren noget. Den skal have sine ordrer i binær form. Enhver ordre må altså kodes i et ord, hvilket normalt betyder en del arbejde.

Dette arbejde overtages af en såkaldt assembler. Den er et program, der oversætter de ordrer, der foreligger i tekstform, til de tilsvarende binære ordreord. Den tekst, som vi oprindelig indtastede, betegnes som mnemonic eller mnemokode. En sådan tekst er virkelig betydeligt lettere at læse, eller siger $4280 Dem måske mere end CLR.L D0?

I dette kapitel vil vi beskrive arbejdet med de forskellige assemblere. Vi vil omtale:

**ASSEM**
Amiga udvidelses-programpakkens assembler. Denne assembler er ret ydedygtig, men bliver i visse forhold klart overgået af de to kolleger.

**Profimat AMIGA**
DATA-BECKERs assembler, der allerede har gjort et godt indtryk i ATARI-ST versionen. Foruden assembleren indeholder den også en såkaldt debugger, som man kan teste, og eventuelt rette færdige programmer med.

**KUMA-SEKA**
Den gennemprøvede assembler, også udstyret med en debugger, der efter min mening er den, der egner sig bedst til at skrive og eksperimentere med øvelsesprogrammer. Det er grunden til, at programmerne her i bogen er skrevet med denne assembler.

Det er fælles for alle assemblere, at de kan oversætte et program, der er skrevet i mnemokode og lagre det som et køreklart program på diskette. For at kunne afprøve det oversatte program direkte, kræves en debugger, hvad den første af de her omtalte assemblere ikke har.

## 3.1. Udvidelses-programpakkens assembler

Denne assembler hører til de rene diskette assemblere. Det betyder, at den

kun kan assemble, hvad der foreligger som tekstfil på diskette og atter skriver resultatet på disketten. Det er altså ikke muligt at foretage direkte indtastninger eller prøvekørsler af det nye program.

ASSEM kaldes fra CLI, idet man indtaster dens navn og derefter de nødvendige parametre.

Kaldet foregår i det enkleste tilfælde således:

**ASSEM kilde -O maal**

hvor "kilde" er navnet på kildefilen med programteksten, og "maal" er navnet på den fil, hvori resultatet af assembleringen skal skrives. "-O" betyder, at det følgende filnavn skal bruges til den objektfil, der skal frembringes.

Der findes flere andre muligheder, for samtidig at angive parametre. De skrives altid sammen med deres option (som -O), for at assembleren kan vide, hvad den skal gøre med den fil, der angives. Disse mulige options, der altid skal følges af et filnavn, er:

**-O** Objektfil (se ovenfor)

**-V** I denne fil skrives alle de meldinger, der resulterer af assembleringen (f.eks. fejlmeldinger). Hvis der ikke er angivet nogen, så fremkommer disse meldinger i CLI-vinduet.

**-L** Udskriften af det assemblerede program bliver ledt over i filen. Man kan f.eks. også angive med PRT:, at der skal printes ud.

**-H** Filen læses ved starten af den fil, der skal assembleres, og bliver assembleret med.

**-E** Filen oprettes, og indeholder de linier, hvor EQU-direktiverne står.

**-C** Denne option anføres uden filnavn, men med endnu en option. Den kan også skrives OPT. Den følges af tegnet:

**OPT S.** Der oprettes en symboltabel, hvor alle labels opføres med deres værdier.

**OPT X.** Der oprettes en liste med krydshenvisninger (hvor skal hvilken label anvendes).

**OPT W.** Herefter skal følge et tal, der fastlægger, hvor meget arbejdslager, der skal reserveres.

Assembleren frembringer så til sidst en såkaldt objektfil, som dog endnu ikke er køreklar. Hvis den skal gøres køreklar, skal man også kalde linkeren ved navn ALINK. Dette program er i stand til at sammenknytte flere assemblerede eller kompilerede objektfiler, og gøre dem til et køreklart program. I det simpleste tilfælde skriver man følgende kommando i CLI:

**ALINK kilde TO maal**

"Kilde" er den af assembleren frembragte objektfil, "maal" er det programnavn, der sluttelig kan startes direkte.

## 3.2. Profimat Amiga.

*Profimat Amiga fra firmaet DATA-BECKER er en vellykket kombination af editor, assembler og debugger, der er let at betjene. Desværre forelå den endelige version af dette program ikke ved afslutningen af bogen, men programmets konstruktør forsikrede mig, at Amiga-versionen vil komme til at fungere på samme måde som ATARI-ST versionen, som jeg allerede har gode erfaringer med.*

Profimat er opdelt i flere vinduer. Et til assembleren, et til editoren, et til debuggeren og nogle hjælpefunktioner. Fremgangsmåden ved konstruktion af et program, er således ret enkel:

1) **Programmet skrives med editoren og lagres på diskette.**
2) **Assembleren startes og oversætter programmet.**
3) **Om nødvendigt startes debuggeren; det oprettede program load'es og testes.**

I debuggeren er der også mulighed for, kun at lade programdele eller endda kun enkelte ordrer, udføre. Da registrenes og flagenes tilstand også vises i stausregistret efter hvert sådant skridt, kan man ret let afprøve de programmer, der vises her i bogen, med denne debugger.

I modsætning til ASSEM behøver man, til hele arbejdet med maskinprogrammer, kun at load'e profimat en enkelt gang, og skal ikke hele tiden skiftevis load'e editor, assembler, linker og debugger.

*En særlig interessant funktion i Profimat bør også omtales: reassembleren. Med denne debugger-funktion kan man forvandle færdige programmer, som man kun har i køreklar stand, til programmets kildetekst, som man så kan viderebearbejde med editoren, eller ændre og derefter assemblere på ny.*

# 3.3. K-SEKA assembleren

Og nu, last but not least, SEKA fra firmaet KUMA, der foruden assembleren indeholder en simpel teksteditor og en debugger. Dette program styres med enkle kommandoer, og er derfor meget let at håndtere. Desuden er det så alsidigt og hurtigt, at det egner sig særdeles godt til afprøvning af små eksempel- og testprogrammer.

Større programmer kan man naturligvis ligeså godt skrive med det, når man har fundet sig til rette med den lovligt enkle editor.

Lad os begynde med at betragte editoren.

Hvis man vil load'e et program som kildekode (tekst) i editoren, så indtaster man ganske enkelt "r" (read). Der følger en forespørgsel "FILENAME>" om hvilket filnavn, der skal load'es, hvorpå man skriver navnet på tekstfilen. Hvis man også har oprettet denne fil med SEKA, så bærer filen på disketten det valgte navn med tilføjelsen "S". For at load'e filen behøver man ikke at tilføje .S, da det sker automatisk. S'et står for "source", hvad der på dansk betyder kilde.

Lagringen af et oprettet eller modificeret program, sker med kommandoen "w", som igen spørger efter navnet.

Hvis man her indtaster "Test", lægges filen "Test.S" på disketten. Det er en normal tekstfil i ASCII-format.

Til indlæsning eller ændring af et program, byder SEKA nu to muligheder:

En simpel linie- og en skærmbilled-editor. Den sidste kommer man ind i, ved at trykke på escape-tasten. Den øverste skærmbilled- eller vinduesdel bliver reserveret til editoren, hvori man kan køre rundt med cursoren og foretage ændringer i teksten. Herunder bliver enkelte tegn pricipielt skubbet ind i den bestående tekst og nye linier automatisk forsynet med nummer. Med endnu et tryk forlader man skærmbilled-editoren.

Det er egentlig ikke nødvendigt, at sige ret meget mere om denne editor. Den er kun egnet til simple indtastninger og ændringer. De øvrige funktioner indtastes atter i den normale kommando-funktionsmåde, hvor ">"-tegnet tjener som opfordring til indtastning eller "klartegn".

De følgende kommandoer er til rådighed for tekstbearbejdning. (<n> står for et tal, betydningen af kommandoen finder De i den runde parenteser.)

| Kommando (betydning) | Funktion |
|---|---|
| t (target) | Sætter cursoren på tekstens øverste linie. |
| t <n> | Sætter cursoren på den linie, der angives af <n>. |
| b (bottom) | Sætter cursoren på tekstens sidste linie. |
| u (up) | En linie op. |
| u <n> | <n> linier op. |
| d (down) | En linie ned |
| d <n> | <n> linier ned |
| z (zap) | Sletter den aktuelle linie. |
| z <n> | Sletter n linier fra cursorpositionen. |
| e (edit) | Tillader editering af den aktuelle linie (og kun den). |
| e <n> | Editering af linie <n>. |
| ftext (find) | Søger fra den aktuelle linie efter den angivne tekst. Herunder tages hensyn til store og små bogstaver, som derfor skal indtastes korrekt. Tomme mellemrum, der indtastes efter "f", søges der også efter! |
| f | Søger videre efter den tidligere opgivne tekst. |
| i (insert) | Starter linieeditoren. Nu kan man indtaste et program, linie for linie, men uden mulighed for at hoppe til en anden linie med cursoren. Linienumre genereres automatisk, og efterfølgende linier bliver ikke slettet, men skubbes videre. |
| ks (kill source) | Efter et kontrolspørgsmål (Sure?), bliver kildeteksten slettet, hvis man svarer "y". Ellers sker der ingenting. |
| o (old) | Ophæver "ks"-funktionen og redder således den gamle tekst. |
| p (print) | Udlæser den aktuelle linie. |
| p <n> | Udlæser <n> linier fra cursorlinien af. |

Det var K-SEKA's editorfunktioner. I forbindelse med skærmbilled-editoren udgør de en ret nem måde at forarbejde tekst på. Således kan man, når man arbejder med skærmbilled-editoren, slette den aktuelle linie og eventuelt også andre linier, ved at gå til kommando-funktionsmåde med escape og direkte indtaste "z" eller "z <n>".

Man kan gå frem på følgende måde, når man vil editere alle de linier, som f.eks. en "trap"-ordre indeholder:

— **gå til tekstens begyndelse med "t",**

— **søg den første linie med en "trap"-ordre ved hjælp af "ftrap",**

— **tryk escape og editer linien,**

— **hop tilbage til kommandofunktionsmåde med et nyt escape,**

— **søg videre med "f", escape, osv. til tekstafslutningen nås.**

Hvad der måske lyder noget omstændeligt, viser sig i praksis at være helt igennem anvendeligt og hurtigt. Det er bedst at eksperimentere lidt med editeringsmulighederne, for at komme til at betjene editoren sikkert.

Nu følger nogle kommandoer, der er beregnet på arbejdet med disketter:

| Kommando (betydning) | Funktion |
|---|---|
| **v (view files)** | **Udlæs indholdsfortegnelsen for en diskette. Der kan samtidig angives en anden diskettestation eller et subdirectory, f.eks. bevirker "vc" en listning af "c"-subdiretory'et og gør samtidig dette til det aktuelle directory.** |
| **kf (kill file)** | **Der spørges efter et filnavn. Filen bliver så slettet (uden kontrolspørgsmål).** |
| **r (read)** | **Efter indtastningen af kommandoen bliver der spurgt efter filnavnet (FILENAME>), der skal load'es, og denne fil bliver så load'et. Indtastes kun "r", så indlæses en tekstfil i editoren.** |
| **ri (read image)** | **Load'er en fil i arbejdslageret. Efter indtastning af filnavnet spørger SEKA om adressen (BEGIN>), hvorfra der skal load'es, og derefter om den slutadresse (END>), hvortil der maksimalt skal load'es.** |

| Kommando (betydning) | Funktion |
|---|---|
| **rx (read from auxiliary)** | **Svarer til ri-funktionen, blot læses ikke fra diskette, men fra den serielle port (filnavnet bortfalder derfor).** |
| **rl (read linkfile)** | **Tillader indlæsningen af en linkfil, der er oprettet med SEKA (forklaring følger senere). Der stilles først et kontrolspørgsmål, idet tekstlageret slettes ved load'ning af en linkfil.** |
| **w (write)** | **Der følger et spørgsmål om det filnavn, hvorunder den aktuelle tekst skal lagres. Navnet gives automatisk filnavnudvidelsen .S, hvorpå man kan kende SEKA-filer.** |
| **wi (write image)** | **Lagrer et område på diskette efter indtastning af navn, begyndelse og slutning.** |
| **wx (write to auxiliary)** | **Som "wi", blot udlæses der via det serielle interface.** |
| **wl (write linkfile)** | **Lagrer efter at have spurgt om navnet på en linkfil, der er assemblet med "l"-option'en. Hvis den ikke er til stede, fremkommer meldingen "** Link option not specified".** |

Hvis man altså har indtastet, eller load'et et program, kan man nu kalde assembleren for at oversætte programmet. Herunder er det nok at indtaste "a", så følger et spørgsmål om option'en. Indtastes kun return, så assembleres normalt, dvs. programmet bliver oversat fra lageret til lageret uden udlæsning af resultatet. Resultatet kan derefter aflæses direkte.

Man kan også indtaste en eller flere af følgende options:

v bevirker udlæsning af resultatet på skærmen.

p eller

e dirigerer med en titellinie udlæsningen over på en printer.

h får udlæsningen til at stoppe, efter hver side og venter på, at der trykkes på en tast. Det er ret nyttigt til at kontrollere udprintningen på skærmen, eller sætte et nyt ark papir i printeren.

**o** lader assembleren optimere alle mulige branch-ordrer. Herunder sættes, om muligt, et .S på branch-ordren, hvad der får programkoden til at blive kortere. Der optræder så nogle meldinger, der dog kan ignoreres.

**l** bevirker frembringelse af en kode, der kan linkes, og kan lagres med "wl" eller "rl".

I tilslutning til selve listningen udprintes en symboltabel (om ønsket). I denne tabel udlæses alle labels med deres værdier samt macro-navne. En macro tillader, at man sammenfatter flere ordrer til en enkelt. Eksempelvis:

De har skrevet en rutine ved navn "pline", der udlæser en tekst, som register A0 peger på. Derfor skal De hver gang skrive:

```
lea text,a0     ;pointer på tekst i A0
bsr pline       ;udlæs tekst
```

Det kan forenkles ved, at der defineres en makro til funktionen. For at opnå det, skriver man i begyndelsen af programmet:

```
print: macro    ;macro ved navn 'print'
  lea ?1,a0     ;parametre i A0
  bsr pmsg      ;udlæs tekst
endm            ;slut på macro
```

I programmet kan man nu blot skrive:

```
print   text    ;udlæs tekst
```

Linien erstatter i virkeligheden assemblere med macro'en. Parameteren "tekst" indføjes så der, hvor der står "?1". Man kan overgive flere parametre til en macro, som man så kan benytte med "?2" osv.

Nu kan man vælge, om man vil se macro'erne i assemblerens udlæsningslistning. Dette og meget mere bliver indføjet i programteksten ved hjælp af kommandoer til assembleren, pseudo-operatorer (kort pseudoops). SEKA kender følgende pseudoops:

**dc** definerer een eller flere data, der skal stå på dette sted i programmet. Herunder angives ordlængden af .B, .W eller .L. Hvis de udelades, antages længden at være .B. Tekster kan ligeledes indtastes, omgivet af halve anførelsestegn (') eller rigtige anførelsestegn.

Eksempel: dc.b "Hallo", 10, 13, 0

**blk** Reserverer et antal bytes, ord eller lange ord, alt efter hvad der angives (.B, .W, .L). Den første parameter angiver antallet af ord, som skal reserveres. Den anden, der er valgfri, anvendes til fyldning af lagerområdet.

Eksempel: blk.w 10,0

**org** Den efterfølgende parameter angiver den lageradresse, hvori det (absolutte) program skal assembleres.

Eksempel: org $40000

**code** Omstiller igen til den relative funktionsmåde, hvor et program assembleres sådan, at det begynder på adresse 0. Omadresseringen, alt efter hvor programmet er load'et, tager Amigaen sig selv af.

**data** Herefter følger kun data, ikke flere programmer (kan bortfalde).

**even** Gør den aktuelle adresse lige, ved at tilføje en udfyldningsbyte.

**odd** Modstykket til "even", gør adressen ulige.

**end** Afslutter assembleringen.

**equ** Tildeler af værdier til labels.
eller =
Eksempel: vaerdi = 123, resp. vaerdi equ 123

**list** Omstiller igen til udlæsning (efter "nlist"). Følgende parametre kan angives med og bevirker udlæsning af:

c macro-kald
d macro-definitioner
e macro-udvidelser af programmet
x kode-udvidelser

Eksempel: list e

**nlist** Frakobler udlæsningen. Man kan angive de samme parametre, som ved "list".

**page** Bevirker en sidefremføring på printeren, resp. at der begyndes på en ny udprintningsside.

**if** Den følgende parameter afgør, om der skal assembleres videre: er den nul assembleres der ikke mere.

**else** Hvis "if"-parameteren er nul, så assembleres der videre.

**endif** Slut på den betingede assemblering.

**macro** Start på en macro-definition.

**endm** Slut på en macro-definition.

**?n** Lader den n'te parameter fra den linie, der kaldes, indføjes i en macro.

**?0** Genererer for hvert macro-kald et nyt trecifret tal, der er meget nyttigt til lokale labels.

Eksempel: x?0: bsr psmg

**illegal** Frembringer en ulovlig maskinordre.

**globl** Definerer de efterfølgende labels som globale, når assemblerens "l"-option er valgt.

Når man til slut har assembleret sit program, ligger den kaldte programkode i lageret. Med kommandoen "h" er det muligt at se, hvor stort programmet er, og hvor i lageret det findes. Herunder udlæses begyndelses- og slutadresser hexadecimalt og længden decimalt (alt efter de sidst udførte operationer).

**Work** Arbejdslager, der er defineret ved starten.

**Src** Tekst i lageret.

**Relc** Programmets relokationstabel.

**RelD** Dataområdets relokationstabel.

**Code** Frembragt programkode.

**Data** Programmets dataområde.

Man finder altså programmet på den adresse i lageret, der er anført under Code. Når man vil starte programmet, er det besværligt altid at skulle opgive denne adresse. Det anbefales derfor at afmærke programmets begyndelse med en label (f.eks. "run:"). Så kan man starte programmet med "g"-ordren på følgende måde:

**g run**

"g"-ordren (go) hører til SEKA's debugger-kommandoer. Her er en oversigt:

**x** Udlæsning af alle registre.

**xr** Udlæsning og ændring af et register (f.eks. xd0).

**gn** Hop til adressen n. Derpå spørges om breakpoints, dvs. adresser, hvor programmet skal afbrydes.

**jn** Som ovenfor, dog hoppes med "JSR" til programmet, som skal afsluttes med "RTS".

**gn** Udlæsning af lagerindholdet fra adresse n. Desuden kan ordstørrelsen vises med.

Eksempel: q.w $10000

**nn** Disassembleret udlæsning fra adresse n.

**an** Direkte assemblering fra adresse n. Der indlæses så direkte programordrer.

**mn** Modifikation af lagerindholdet fra adresse n. Også her kan man angive ordstørrelsen. Indtastningen afbrydes med escape.

**sn** Udførelse af programordren, som PC peger på. Ved angivelse af n udførelse af n programtrin.

**f** Opfyldning af et lagerområde (valgfri angivelse af ordbredde). Der spørges separat om alle enkelte parametre.

**c** Kopiering af et lagerområde til et andet. Der spørges separat om alle nødvendige parametre.

Udlæsning af værdien af et udtryk eller en label.

Eksempel: ? run+$1000-256

**a** Sætning af en kommandosekvens, som programmet får overgivet ved starten, som om det var startet fra CLI'en med denne sekvens.

**!** Udgang af SEKA-assembleren efter kontrolspørgsmål.

De regneoperationer, som er anført i eksemplet med "?", viser, at SEKA også kan regne. Det kan man også anvende i programmer. Følgende operationer fungerer i SEKA:

| | |
|---|---|
| + | addition |
| - | subtraktion |
| * | multiplikation |
| / | division |
| & | logisk OG |
| ! | logisk ELLER |
| ~ | eksklusivt eller (EXOR) |

Operationerne kan kombineres. Endnu en forskel ses i valget af det anvendte talsystem. $ betyder hexadecimalt, @ oktalt og % binært; hvis der ikke angives et af disse symboler, antages at det er et decimaltal.

Men tilbage til debuggeren. Som nævnt spørges efter indtastningen af "g adresse", efter såkaldte breakpoints. Der kan indtastes indtil 16 adresser, på hvilke programudførelsen vil standse. Hvis man ikke angiver noget breakpoint, men trykker return skal programmet slutte med en ILLEGAL-ordre. Hvis derimod sluttes med et RTS, så hentes en tilfældig returhop-adresse fra stacken, og der hoppes derhen. Det er ganske vist for det meste adressee 4, hvorfra SEKA returnerer med meldingen "*** illegal instruction at $000004", men det kan også være en helt anden, hvorefter computeren igen kan give sig til at meditere...

På de steder, der er opgivet som breakpoints, sætter SEKA en ILLEGAL-ordre, efter at den har gemt det gamle indhold af det pågældende lagerord. Hvis processoren under udførelsen af programmet kommer til en sådan ordre, så hoppes via illegal-instruction-vektoren, der på forhånd er indstillet således af SEKA, til debuggeren. SEKA reparerer igen de forandrede lagerceller, og printer så statuslinien ud, hvoraf fremgår, at programmet er afbrudt.

Teknikken med breakpoints er meget fordelagtig, når man vil søge en fejl i et program. Man kan f.eks. sætte et brekapoint før en tvivlsom rutine, og starte programmet. Når det så igen afbrydes, på det pågældende sted, kan man gennemkøre funktionen trin for trin ved hjælp af enkelttrin-funktionen "s" og betragte resultatet af hver enkelt ordre med statuslinien, og således komme på sporet af den pågældende fejl.

*Sådanne programfejl, der så lumsk sniger sig ind i programmerne, kalder man også "bugs" (lus). Derfor hedder et program til aflusning af andre programmer netop også en debugger.*

# 4. De første programmer

Nu er vi så vidt, at vi skal afprøve vor nye viden på programmer, der ikke kun tjener til demonstration, men også opfylder et formål. Jeg går her ud fra, at De har en SEKA-assembler og har load"et den, og udelader linienumrene i programeksemplerne.

Hvis der anvendes en anden assembler, er der naturligvis visse ting, der skal håndteres på en anden måde end angivet. Disse forskelle er allerede omtalt i kapitlet om de forskellige assemblere.

For at kunne afprøve programeksemplerne direkte, og for det andet at kunne dem igen senere, er de udført som underprogrammer. Man kan derfor efter assemblering og eventuel tildeling af de ønskede værdier til de respektive registre, starte programmet ved at skrive "j programnavn", og bagefter aflæse resultatet, eller resultaterne, direkte fra registerindholdene, eller med "q adresse" fra lageret.

Lad os begynde med et simpelt eksempel: addition af tal fra en tabel.

## 4.1. Addition af tal

Vi forestiller os, at vi har en række tal liggende i lageret, som vi vil lægge sammen. Vi tager som eksempel fem tal med et ords længde, hvis sum skal skrives i register D0. Den nemmeste metode er:

```
addition1:
        clr.l   D0            ;slet D0 (=0)
        move    tabel,d0      ;første indførelse i D0
        add     tabel+2,d0    ;adder anden indførelse
        add     tabel+4,d0    ;adder tredie indførelse
        add     tabel+6,d0    ;adder fjerde indførelse
        add     tabel+8,d0    ;adder femte indførelse
        rts                   ;tilbage til hovedprogrammet

tabel:          dc.w 2,4,6,8,10
```

Når programmet afprøves med "j addition", vil man se, at dataregister D0 faktisk indeholder summen af værdierne.

Jeg tror, det er let at forestille sig, at det må kunne gøres meget mere elegant og kort. Som motivation til det, kan nævnes det tilfælde, hvor de data, der

skal adderes, skal kunne have variable adresser, dvs., når et og samme program skal kunne addere forskellige tabeller.

Den ovenfor anførte metode vil i så fald svigte, fordi den kun aflæser og adderer ganske bestemte adresser. Men hvortil har vi i Amiga'en en processor med så mange forskelligartede adresseringsmåder?

Jeg vil foreslå, at vi overgiver adressen på tabellen til underprogrammet i et adresseregister (f.eks. A0). Det register kan så anvendes direkte, som en pointer på tabellen, ved hjælp af indirekte adressering. Vi kan altså i programmet ændre udtrykket "tabel+x" til "2*x(a0)".

```
addition1:
    clr.l  D0           ;slet D0 (=0)
    move   0(a0),d0     ;første indførelse i D0
    add    2(a0),d0     ;adder anden indførelse
    add    4(a0),d0     ;adder tredie indførelse
    add    6(a0)d0      ;adder fjerde indførelse
    add    8(a0),d0     ;adder femte indførelse
    rts                 ;tilbage til hovedprogrammet

tabel:     dc.w 2,4,6,8,10
```

Også dette program adderer de fem tal korrekt. Grunden til, at der tages trin på 2 i angivelsen af offsets, ligger i, at vi jo tilgår ord, der hver er 2 bytes lange.

Yderligere "forskønnelse" af programmet vil være at anvende udtrykket "(a0)+" i stedet for "x(a0)". Derved bliver adresseregister A0 automatisk blive talt op for hver tilgang til tabellen, og med det antal bytes, der blev udlæst på een gang (her altså 2). Forskellen fra den ovenstående variant er ganske vist den, at indholdet af registret, så bliver forandret. Pointeren vil pege på den næste ubenyttede byte, resp. ord i lageret.

Endnu et problem. Antallet af bytes, som skal adderes, skal også være variabelt og f.eks. overføres til register D1. Vi må altså, finde på en anden måde at programmere på, da det ikke lader sig gøre med den oprindelige metode.

Nu kommer begrebet løkke ind i sagen. Vi skal addere det næste ord D1 gange. Vi kan atter anvende (a0)+ som adresseringsmetode (adresseregister indirekte med postinkrement), da det automatisk giver den til enhver tid følgende værdi.

Så til løkken. Den realiseres på den måde, at efter hver addition formindskes indholdet af tælleren, i dette tilfælde register D1, med 1. Hvis resultatet er nul, så er arbejdet afsluttet, ellers udføres en addition til. Det ser sådan ud i programmet:

```
addition2:
       clr.l  d0              ;slet D0
loop:                         ;label til løkkeindgang
       add    (a0)+,d0        ;opadder værdi
       subq   #1,d1           ;formindsk tæller
       bne    loop            ;fortsæt hvis ikke nul
       rts                    ;ellers slut

tabel:        dc.w 2,4,6,8,10
```

Nu sættes pointeren på tabellens adresse med "x a0" og tælleren på 5 med "x d1". Herefter kan programmet startes med "j addition2". Man kan så afprøve det med en anden værdi til tælleren (f.eks. 3) og sammenligne med det forventede resultat.

Her er en sidste afpudsning for at gøre programmet mere elegant og hurtigt. Vi anvender til det formål MC68000's specielt på løkker beregnede ordre "DBRA". Denne ordre klarer nemlig to trin, som vi har anvendt to ordrer til i vort program: nedtælling af tælleren og den betingede forgrening. Denne forgrening sker ganske vist lige så længe, som tælleren er positiv. Således forgrenes der altså også ved nul, så at vort program ikke længere fungerer på nøjagtig samme måde som tidligere. Vi skal derfor blot overføre antallet af værdier -1 til tælleren, her altså 4.

```
addition2:
       clr.l  d0              ;slet D0
loop:                         ;label til løkkeindgang
       add    (a0)+,d0        ;opadder værdi
       dbra   d1,loop         ;tæl D1 ned, forgren hvis positiv
       rts                    ;ellers slut

tabel:        dc.w 2,4,6,8,10
```

Ved dette program vil det være naturligt at studere udførelsen nærmere. Derfor load'er vi igen pointeren A0 med "x" og tælleren D1 med de nødvendige værdier, og indstiller så med "x pc" programtælleren på "addition2". Nu kan vi ved hjælp af "s" lade udføre eet programtrin ad gangen, og se på resultatet. Ganske vist skal man ved afslutningen af programmet, når ordren RTS vises, ikke lade denne udføre, da der ellers bliver hentet en returhopadresse fra stacken, der slet ikke er blevet lagt ind på den.

Ved afslutningen af eksemplet endnu en lille opgave, som læseren selv kan udføre: skriv programmet om på en sådan måde, at det adderer nogle bytes eller lange ord. Og prøv så, at skrive et program, der subtraherer alle følgende værdier fra den første værdi i en tabel, altså med tabellen

```
tabel:      dc.w   50,8,4,6
```

giver resultatet af 50-8-4-6 = 32 ($20).

## 4.2. Sortering af en tabel

Lad os blive ved bearbejdningen af tabeller. Nu vil vi ikke blot udlæse dataene, men også ændre dem. Tabellen skal nemlig sorteres i opadgående retning.

Lad os først overveje, hvordan vi kan håndtere sorteringen. Den nemmeste metode er følgende:

Man sammenligner den første med den anden værdi. Hvis anden værdi er større end den første, så er det i orden, og man går over til næste trin, hvor man sammenligner den anden med den tredje værdi osv. Hvis man er kommet til det sidste par, og den foregående værdi hver gang var mindre end den efterfølgende, så er sorteringen afsluttet (resp. unødvendig).

Hvis man finder et par værdier, hvor den anden er mindre end den første, så ombyttes disse to værdier ganske enkelt. For at det ikke skal glemmes, sætter man et eller andet mærke (her fortrinsvis et register), der afprøves i slutningen af tabellen. Hvis det er sat, så er tabellen sandsynligvis ikke færdigsorteret: man sletter altså mærket igen og begynder forfra. Hvis mærket stadigvæk er slettet ved afslutningen, så er sorteringen afsluttet.

Vi opbygger et program på denne måde. Først bestemmer vi de nødvendige variabler, og tildeler dem registre. Vi skal altså bruge en pointer på den tabel, der skal sorteres (A0), en tæller (D0) og et mærke (D1). Da vi hele tiden ændrer værdierne i løbet af programmet, indretter vi tillige to registre, A1 og D2, hvor vi midlertidigt kan lagre begyndelsesværdierne (adresse og antal tabelindførelser).

Nu begynder vi at opbygge programmet. Vi giver underprogrammet adressen på tabellen i A1 og antallet af værdier i D2.

```
sortering:                    ;programmets indhopadresse
        move.l  a1,a0         ;load'ning af pointer med adresse
        move.l  d2,d0         ;antal i tælleren
        subq    #2,d0         ;korrektion af tællerværdi
        clr     d1            ;sletning af mærke
```

Såvidt forberelserne. Pointeren og tælleren er forberedt, og mærket er slettet. Tælleren er formindsket med 2, fordi vi vil arbejde med DBRA-ordren (1 fratrækkes) og med X tal kun kan udføre X-1 sammenligninger (1 fratrækkes igen).

Derefter følger løkken, hvori værdierne sammenlignes. Vi sammenligner, ord for ord, en værdi med den følgende. Det ser således ud:

```
loop:
        move   2(a0),d3      ;næste værdi i register D3
        cmp    (a0),d3       ;sammenlign værdierne
```

Omvejen over register D3 er desværre nødvendig, fordi det ikke er muligt med CMP (A0),2(A0). Hvis den første værdi er større eller lig den første, kan vi udelade ombytningen, altså:

```
        bcc    noswap        ;forgren, hvis større eller lig
```

Nu ombyttes der i givet fald (det går desværre ikke med "exc 2(a0),(a0), da den form for adressering ikke eksisterer):

```
swap:
        move   (a0),d1       ;gem den første værdi
        move   2(a0),(a0)    ;kopier anden værdi ind i første
        move   d1,2(a0)      ;og læg første ind i anden
        moveq  #1,d1         ;sæt maerke
noswap:
```

Nu skal pointeren forhøjes, og der fortsættes med næste par osv., indtil tælleren bliver negativ:

```
        addq.l #2,a0         ;pointer+2
        dbra   d0,loop       ;fortsæt løkke til afslutning
```

Vi ser efter, om mærket er sat, og begynder forfra, hvis det ikke er 0:

```
        tst    d1            ;test mærke
        bne    sortering     ;ikke færdigsorteret!
        rts                  ;ellers færdig: returnerer...
```

Hvis mærket er slettet, så er vi færdige, og underprogrammet er afsluttet. Der vendes tilbage til hovedprogrammet med RTS.

Her er programmet i samlet form:

```
sortering:                          ;programmets indhopadresse
        move.l  a1,a0               ;load'ning af pointer med adresse
        move.l  d2,d0               ;antal i tælleren
        subq    #2,d0               ;korrektion af tællerværdi
        clr     d1                  ;slet mærke
loop:
        move    2(a0),d3            ;næste værdi til register D3
        cmp     (a0),d3             ;sammenlign værdier
        bcc     noswap              ;forgren hvis større eller lig
swap:
        move    (a0),d1             ;gem første værdi
        move    2(a0),(a0)          ;kopier anden værdi ind i første
        move    d1,2(a0)            ;og læg første i anden
        moveq   #1,d1               ;sæt mærke
noswap:
        addq.l  #2,a0               ;pointer+2
        dbra    d0,loop             ;fortsæt løkke til afslutning
        tst     d1                  ;test mærke
        bne     sortering           ;ikke færdigsorteret!
        rts                         ;ellers færdig, returner...
```

Her er der altså blevet brugt nogle registre til at lagre aktuelle værider. Det forekommer dog ofte, at et underprogram ikke må ændre nogen, eller i alt fald kun bestemte, registre. Dertil findes en maskinordre, som med et slag kan lægger flere registre på stacken og dermed gemmer dem. Hermed menes ordren MOVEM, der betyder "move multiple". Lad os indføje ordren i vort program, naturligvis to gange, da de gemte værdier også skal hentes fra stacken igen. Til det behøver vi endnu en label, som vi kalder start, og hvor underprogrammet bliver startet:

```
start:
        movem.l  d0-d7/a0-a6,-(sp)  ;gem register
sortering:
        osv.
        ...
        ...
        bne      sortering          ;endnu ikke færdig
        movem.l  (sp)+,d0-d7/a0-a6  ;gentag register
        rts                         ;og færdig
```

Denne mægtige ordre forskyder altså flere registre på een gang. Samtidig kan man angive, hvilke registre, der skal forskydes. Hvis man f.eks. kun vil forskyde registrene D1,D2,D3,D7,A2 og A3, så skriver man

```
movem.l   d1-d3/d7/a2-a3,-(sp)
```

Før vi afslutter temaet sortering, er igen en lille opgave: forsøg engang at forandre programmet således, at det sorterer omvendt. Altså bringer det største tal foran og det mindste sidst.

## 4.3. Forvandling af talsystemer

Som man sikkert har bemærket i kapitlet om talsystemer, er det besværligt at omregne fra det ene talsystem til det andet. Desuden er der endnu en form for forvandling af tal: fra eller til tegnkæder, der kan indtastes fra tastaturet, eller udlæses på skærmen.

*Vi vil nu se nærmere på nogle af denne store mængde mulige forvandlinger, og formulere dem som programmer. Vi begynder med forvandlingen af binære tal til en tegnstreng, der udtrykker tallets værdi hexadecimalt.*

### 4.3.1. Binære til hexadecimale tal

Først skal begyndelses- og slutbetingelserne overvejes. Lad os eksempelvis fastlægge dem sådan, at indholdet af dataregister D1 skal belægges med otte pladser, altså som et langt ord, i et bestemt lagerområde som ASCII-tegn, og skal senere kunne udlæses herfra.

Fordelen ved hexadecimal frem for den decimale skrivemåde viser sig netop meget tydeligt ved denne problemstilling. For at få et hexadecimalt ciffer ud af et lagerord, behøver vi kun at tage de tilsvarende fire bits, altså en halv byte, ud og bearbejde dem. Hver halve bytes, der også kaldes en nibble, indeholder præcis et ciffer af det hexadecimale tal!

Lad os nu betragte bearbejdningen af en halvbyte, som vi antager ligger i D2. For at få et tegn, der kan printes ud, må den forvandles til ASCII-kodetegn. Koden er for de 16 mulige tegn i et hexedecimalt ciffer:

| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **A** | **B** | **C** | **D** | **E** | **F** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **$30** | **$31** | **$32** | **$33** | **$34** | **$35** | **$36** | **$37** | **$38** | **$39** | **$41** | **$42** | **$43** | **$44** | **$45** | **$46** |

For at forvandle cifrene 0-9, behøver man altså kun at addere $30. Ved bogstaverne A-F, der har værdierne 10-15, skal der derimod adderes $37. Vort program til vurdering af en halvbyte, skal derfor først skelne mellem, om den halve bytes værdi ligger mellem 0 og 9 eller mellem A og F, og alt efter denne skelnen, enten addere $30 eller $37.

Det omsættes nu til et maskinprogram, og det som underrutine, da det skal kunne kaldes for hver plads i det planlagte lange ord.

```
nibble:
        and     #$0f,d2     ;udskil den nederste nibble
        add     #$30,d2     ;adder straks $30
        cmp     #$3a,d2     ;var det et ciffer?
        bcs     ok          ;hvis ja: færdig
        add     #7,d2       ;ellers adder yderligere 7
ok:
        rts                 ;færdig
```

Rutinen forvandler altså den nibble, der indeholdes i D2 til et ASCII-tegn, som svarer til nibble'ns hexadecimale værdi. For at forvandle en hel byte, må vi kalde rutinen to gange. Hertil et program der, som parameter i A0, får adressen på en buffer, hvor tegnet skal afleveres, og i D1 får den byte, der skal forvandles.

```
        ...
        lea     puffer,a0   ;viser på buffer
        move    #$4a,d1     ;byte der skal forvandles (eksempel)
        bsr     byte        ;og forvandl
        ...

byte:
        move    d1,d2       ;bring værdi til D2
        lsr     #4,d2       ;bring øvre nibble til underste
        bsr     nibble      ;forvandl D2
        move.b d2,(a0)+     ;læg tegn i buffer
        move    d1,d2       ;værdi i D2
        bsr     nibble      ;forvandl laveste nibble
        move.b d2,(a0)+     ;og læg den i buffer
        rts                 ;færdig

nibble:
        ...                 ;program se ovenfor
puffer:         blk.b 9,0   ;buffer til langord-data
```

Først bliver den værdi, der skal forvandles, bragt til D2. Derefter forskydes registeret fire gange til højre, for at transportere den højeste nibble til de laveste 4 bits. Derigennem bliver, efter det underprogramkald der nu følger, HI-nibble'n først bragt til bufferen med "move.b d2,(a0)+". Derefter bringes den oprindelige byte igen til D2 og forvandles, hvad der til sidst giver LO-nibble'n som ASCII-tegn i D2, som bliver bragt til bufferens næste byte.

Bufferen er tilstrækkeligt lang til at optage et langt ord som cifre, inklusive en afsluttende nul-byte, der sædvanligvis anvendes af skærmbilledudlæsningsrutiner som tekstafslutningstegn. Skærmbilledudlæsningen venter vi dog med til senere. Lad os først bekymre os om den fuldstændige forvandling af det lange ord.

*Også ved forvandlingen af hele det lange ord må vi passe på, at rækkefølgen af nibbles bliver overholdt. For altså at kunne kalde "nibble"-rutinen for første gang, må vi bringe den højeste af de 8 nibbles i det lange ord ned i de laveste 4 bits, uden at der går noget tabt.*

Dertil er forskydningsordren LSR ikke så velegnet, da de bits, der skubbes ud, går tabt. Her er det bedre med rotationsordrerne som ROR eller ROL, der atter skubber den bit, der blev skubbet ud, ind fra den anden side.

Når vi forskyder det oprindelige langord i D1 4 gange til venstre, bliver de 4 højeste bits skubbet ind i de 4 laveste, hvorfra vi kan udnytte dem med vor "nibble"-rutine, og transportere det opståede ASCII-tegn til bufferen. Det hele gentages 8 gange, så er det lange ord fuldstændigt forvandlet, endda med den uhørte luksus, at register D1 bagefter atter har den oprindelige værdi!

```
hexlong:
        lea     puffer,a0       ;pointer på bufferen
        move    #7,d3           ;nibble-tæller på 8-1
loop:
        rol     #4,d1           ;bring højeste nibble ned
        move    d1,d2           ;skriv i D2
        bsr     nibble          ;og forvandl
        move.b  d2,(a0)+        ;tegn i buffer
        dbra    d3,loop         ;det hele gentages 8 gange
        rts                     ;færdig!

nibble:
        ...                     ;program se ovenfor

puffer:         blk.b 9,0       ;buffer til langord-data
```

Dette program indtastes med nibble-rutinen i SEKA og assembleres. Derefter bør det afprøves med en værdi på $12345678 i D1. Man vil kunne fastlslå, at der må være en fejl i programmet, fordi der i bufferen står cifrene "56785678" i stedet for "12345678". Prøv at finde fejlen!

Er den fundet? Fejlen er en hyppigt forekommende sjuskefejl, kan være vanskelig at indse. Fordi assembleren antager, at rotationsoperationen skal foretages i ordbredde, da angivelsen ".l" mangler, er det altid kun det laveste ord i

D1, der bliver roteret, så de samme værdier bliver leveret to gange. Hvis vi ændrer kommandoen til "rol.l", så arbejder programmet som det skal.

På grund af fejlen, har vi imidlertid også set, hvor let det er at forvandle 4-cifrede hexadecimaltal til ASCII-tegn. Det er kun nødvendigt atter at fjerne ".l" fra "rol"-kommandoen og ændre tælleren fra 7 til 3. Straks får man et 4-cifret hexadecimalord.

Nu en lille opgave: lav programmet om, så der kommer 6-cifrede hexadecimaltal ud af det (D1 skal ikke nødvendigvis forblive uændret...)!

Vi vender os til et andet forvandlingsproblem; forvandlingen til et 4-cifret decimaltal.

## 4.3.2. Binære til decimale tal

Som man let kan indse, er dette ikke så let som forvandlingen til et hexadecimaltal. Her kan man nemlig ikke så nemt få fat på de enkelt cifre, ved at vurdere grupper af bits. Vi må finde en anden metode.

Prøv derfor engang se på, hvordan et decimaltal er opbygget. Hvis der er fire cifre, betyder det første, hvor mange gange 1000 er indeholdt i den samlede værdi. Det følgende ciffer hvor mange gange 100 indeholdes i resten, og så fremdeles.

Når vi har en værdi i et register og deler med 1000, så får vi værdien af det første ciffer i vort decimaltal. Da den tilhørende maskinordre til divison ikke alene giver os resultatet af divisionen, men også resten, er det let at arbejde videre med denne rest. Vi deler nemlig med 100, får værdien af decimaltallets 100'er-ciffer, deler igen resten med 10 og får så til slut værdien af 10'er-cifret, og den nye rest giver os samtidig 1'ercifrets værdi i tallet.

På den måde er det ikke så svært, som man først kunne tro. Her er programmet, som trinvis udfører de ovenfor beskrevne skridt, og bringer værdien af D1 i bufferen:

```
deci_4:        ;underprogram til 4-cifrede decimaltal
      lea      puffer,a0     ;pointer på buffer
      divu     #1000,d1      ;deling med 1000
      bsr      ciffer        ;benyt resultat og send rest ned
      divu     #100,d1       ;deling med 100
      bsr      ciffer        ;benyt resultat og send rest ned
      divu     #10,d1        ;deling med 10
      bsr      ciffer        ;benyt resultat og send rest ned
                             ;og benyt rest direkte
```

```
ciffer:
    add     #$30,d1      ;forvandl resultat til ASCII
    move.b d1,(a0)+      ;og bring til buffer
    clr     d1           ;slet laveste ord
    swap    d1           ; og bring rest ned
    rts                  ;og tilbage

buffer:     blk.b  5,0   ;buffer til ASCII-tegn
```

I programmet kan man se et lille trick, der er almindelig anvendt: underrutinen "ciffer", der bliver kaldt tre gange fra underrutinen "deci_4", bliver pludselig til en del af "deci_4"-rutinen. Efter det tredie kald af "ciffer", arbejdes nemlig videre i "ciffer", uden at der har fundet et BSR eller et JSR sted. Hvis processoren støder på et RTS, vendes tilbage fra underrutinen. Denne gang ikke til "deci_4" underprogrammet, men til hovedprogrammet! På den måde sparer man det fjerde "bsr ciffer" og "rts" til afslutning af "deci-4"-rutinen.

Læseren bør nu gennemprøve dette program. Herunder skal man huske, at skrive en værdi i D1, der er mindre end 9999, da der ellers kan ske det mest besynderlige...

Og nu vil vi forsøge det modsatte af den ovenfor stående forvandling: tegnstrenge til binære tal.

## 4.3.3. Hexadecimale til binære tal

Når man ser nærmere på en tegnstreng, der repræsenterer et hexadecimalt tal, udgør hvert af tegnene igen en halvbyte. Man skal altså skrive et program, der arbejder stik modsat det, der forvandler et binært tal til et hexadecimalt.

Her er to forskellige muligheder:

1. **Antallet af hexadecimale cifre er kendt, eller**
2. **antallet er ukendt**

Det første tilfælde er let at programmere, men har den mangel, at man for at kunne indtaste et fircifret hexadecimaltal, også bliver nødt til at skrive værdien 1 med fire cifre, altså som 0001. Det er ikke praktisk, så vi vælger med det samme, det andet tilfælde.

Begynd med forvandlingen af et enkelt ciffer. Forudsæt, at vi overgiver en pointer, på dette ciffer, i adresseregister A0 til forvandling i et underprogram, og vil have den resulterende binære værdi tilbage i dataregister D0.

Programmet ser så sådan ud:

```
nibbleen:                           ;*forvandling af nibble'n fra (A0)
      clr.l   d0                    ;slet D0
      move.b (a0)+,d0               ;hent ciffer, forhøj A0
      sub     #'A',d0               ;subtraher $41
      bcc     ischar                ;ingen overførelse: det var A-F
      add     #7,d0                 ;ellers korriger værdi
ischar:
      add     #10,d0                ;korriger værdi
      rts
```

Lad os følge programmets forløb, ved hjælp af et eksempel. Vi antager, at der i den lagercelle, som A0 peger på, står tegnet "B", der er repræsenteret af ASCII-værdien $42. Tallet load'es først i D0, efter at dette register er slettet.

Ved subtraktion af værdien $41, bliver det til værdien $1, og hermed er vi allerede færdige. Før returneringen til hovedprogrammet lægges nemlig 10 til, hvorved vi kommer til den rigtige værdi 11, altså $B.

Havde vi overgivet et ciffer, så ville der ved subtraktionen, være fremkommet et overløb til det negative, og dermed ville C-flaget være blevet sat. Som det andet eksempel tages cifret 5.

ASCII-værdien for 5 er $35. Ved subtraktionen fratrækkes $41, hvad der giver -12 og sætter C-flaget. Er det tilfældet, så forgrenes der ikke ved BCC-ordren, men der adderes derefter værdien 7, hvorved vi kommer til -5. Så adderes 10, og vi får 5: færdig!

*Rutinen har en mangel. Hvis der overgives et forkert tegn, der ikke repræsenterer noget hexadecimalciffer, så returneres et eller andet vanvittigt resultat. Men lad os først glemme fejlindtastninger.*

Vi går videre til flercifrede hexdecimaltal. Det første ciffer, som vi forvandler, har den højeste værdi, og må derfor også repræsenteere den højeste nibble. For at kunne gøre det, og alligevel kunne håndtere vilkårligt lange tal (ikke helt vilkårlige, tallene skal passe ind i et langt ord, altså have maksimalt 8 cifre), anvender vi et trick.

Vi betragter straks hele programmet, som klarer den fuldstændige omregning og returnerer resultatet i D1. Der gås ud fra, at pointeren på tegnstrengen overgives i A0, og at denne tegnstreng afsluttes med en nulbyte.

```
hexin:                    ;*forvandling af et hexadecimaltal
    clr.l   d1            ;slet først D1
hexinloop:
    tst.b   (a0)          ;test ciffer
    beq     hexinok       ;nul, altså færdig
    bsr     nibbleen      ;forvandl ciffer
    lsl.l   #4,d1         ;forskyd resultat
    or.b    d0,d1         ;indføj nibble
    bra     hexinloop     ;og fortsæt
hexinok:
    rts
```

Det omtalte trick består i, at resultatet efter behov bliver skubbet fire pladser, altså en nibble til venstre. Herved forhøjes det sidste ciffers pladsværdi med 16, og der skaffes plads til den nibble, der er blevet leveret af "nibbleen"-rutinen. Hvis programmet støder på nulbyte'n i slutningen af tegnstrengen, bliver det fastslået af TST.B-ordren og programmet afsluttet. Resultatet ligger nu i det lange ord D1.

For at opfange eventuelle fejlindtastninger, må vi foretage visse ændringer i det programmet. Vi laver en tilføjelse på det sted, hvor vi får den værdi tilbage fra "nibbleen"-rutinen, som det aktuelle tegn har antaget.

Hvis værdien i D0 er større end $F, er der en fejl. Det kan vi fastslå på flere måder. Vi vælger den letteste vej, idet vi sammenligner værdien med $10 ved hjælp af en "CMP #$10,D0"-operation. Hvis den er mindre, så er C-flaget sat, idet der har fundet en overførelse sted ved den subtraktion som CMP udfører. Ellers er C slettet, hvad der tyder på en fejl.

Med dette trick kan vi lade testen for en nulbyte falde væk, da den også repræsenterer et ugyldigt tegn. Hele programmet ser dermed således ud:

```
hexin:                    ;*forvandling af et hexadecimaltal
    clr.l   d1            ;slet først D1
hexinloop:
    bsr     nibbleen      ;forvandl ciffer
    cmp     #$10,d0       ;test om gyldig
    bcc     hexinok       ;nej, altså færdig
    lsl.l   #4,d1         ;forskyd resultat
    or.b    d0,d1         ;indføj nibble
    bra     hexinloop     ;og fortsæt
hexinok:
    rts                   ;slut på forvandling
```

```
nibbleen:                    ;*forvandling af nibble'n fra (A0)
    slr.l   d0               ;slet D0
    move.b (a0)+,d0          ;hent ciffer, forhøj A0
    sub     #'A',d0          ;subtraher $41
    bcc     ischar           ;ingen overførelse: det var A-F
    add     #7,d0            ;ellers korriger værdi
ischar:
    add     #10,d0           ;korriger værdi
    rts
```

Det var så metoden, til at forvandle hexadecimaltal til binære tal. Lad os nu gå over til decimaltal, hvad næppe er meget vanskeligere.

## 4.3.4. Decimaltal til binære

Vi kan stort set, anvende samme metode som ved hexadecimltal. Da vi også her går ud fra et ukendt antal cifre, er teknikken ved forhøjelse af den øjeblikkelige pladsværdi ligesådan. Ganske vist kan vi ikke opnå det ved en simpel forskydning, men må multiplicere resultatet med 10 og addere værdien af cifret.

Her er programmet til forvandling af decimaltal:

```
decin:                       ;*forvandling af et decimaltal
    clr.l   d1               ;slet først D1
decinloop:
    bsr     cifferin         ;forvandl ciffer
    cmp     #10,d0           ;test om gyldig
    bcc     decinok          ;nej, altså færdig
    mulu    #10,d1           ;forskyd resultat
    add     d0,d1            ;indføj nibble
    bra     decinloop        ;og fortsæt
decinok:
    rts                      ;slut på forvandling

cifferin:                    ;*forvandling af nibble'n fra (A0)
    clr.l   d0               ;slet D0
    move.b (a0)+,d0          ;hent ciffer, forhøj A0
    sub     #'0',d0          ;subtraher $30
    rts
```

Programmet kan desværre "kun" forvandle tal indtil 655350, i modsætning til hexadecimalforvandlingsrutinen. Det skyldes, at MULU-ordren kun kan

multiplicere 16-bits ord. Den sidste multiplikation, der ville ske for at nå den højeste værdi korrekt, ville være den fra $FFFF*10, altså 65535*10, hvad der giver 655350. Normalt er talområdet helt tilstrækkeligt, så vi kan spare os, at gøre programmet mere kompliceret.

# 5. Hardware-register anvendelse

Der findes en anden måde end biblioteksfunktionerne, til at tilgå visse informationer, eller udløse bestemte hardware-funktioner: *hardware-registrene.* Det er lagerceller, der ligger på en bestemt adresse, der hverken befinder sig i RAM'en eller ROM'en. Det vedører de direkte interfaces mellem processoren og dens periferikomponenter.

Således finder man for hver af komponenterne en hel række hardware-registre, hvorigennem processoren også griber ind, og dermed kan styre grafik, lyd og ind-/udlæsning i almindelighed. Her er et stort spillerum for assembler-programmøren, som vi skal vise nogle eksempler på.

Registrene anvendes i almindelighed byte for byte. Et eksempel herpå findes i næste kapitel.

## 5.1. Aflæsning af specialtaster

Load SEKA-assembleren, eller et lignende monitor-program, og indtast:

**'q $bfec00'**

Der kommer så en listning, byte for byte, af adresserne $BFEC00 til $BFEC7F, hvori der stadig er to bytes, der gentages. De to bytes viser hele tiden tilstanden i to hardware-registre.

Afspejlingen skyldes, at ikke alle adressebits anvendes ved afkodningen af adressen via den komponent, der er tilgået. Derfor er det kun de højeste 2 bytes af adressen og den laveste bit, bit 0, der spiller en rolle for adresseringen af disse registre. Adressen på de to registre lyder omtrent således: $BFECxx, hvor den underste adressebyte, xx, kun indeholder information, om det ønskede register i bit 0, medens bits 1-7 er ligegyldige. Denne ejendommelige adressering kan genfindes i de fleste hardware-registre.

Men lad os se på, hvilke informationer, der indeholdes i registrene. Til den ende betragter vi det andet register $BFEC01, der indeholder $77. Indtast nu

**'q $bfec00'**

og tryk på alternate-tasten umiddelbart efter trykket på return. Som det ses, forvandles indholdet af hver anden byte ($BFEC01, $BFEC03 osv.) til $37. Det viser den aktiverede specialtasts tilstand. Det samme gælder for de andre specialtaster. Således giver følgende taster disse bytes:

| | |
|---|---|
| Shift til højre | **$3F** |
| Shift til venstre | **$3D** |
| Control | **$39** |
| Alternate | **$37** |
| Amiga til venstre | **$33** |
| Amiga til højre | **$31** |

Man kan altså hele tiden teste i et kørende maskinprogram, om der er trykket på en af disse taster, og i givet fald udløse eller afslutte forskellige funktioner. En progamdel kan f.eks. se sådan ud:

```
staster = $bfec01

    ...
    cmp.b   #$37,staster   ;er der trykket på alternate?
    beq     funktion1      ;ja!
    cmp.b   #$31,staster   ;eller Amiga-tasten til højre?
    beq     funktion 2     ;ja!
    ...                    ;hverken eller...
```

## 5.2. Hvordan man fastslår den løbende tid

Amigaen råder over et indbygget ur, som registrerer klokkeslet og datoen. Uret kan man aflæse med en biblioteksfunktion. Tidsangivelsen er dog ikke praktisk ved alle programmer.

Vil man i et program fastslå, hvor længe der er gået mellem to begivenheder, bliver det ret upræcist, når klokkeslettet anvendes. Det er mere interessant med en løbende tid, som ligger i et hardware-register og tæller hurtigt, men præcist opad. Sådan en tidsangiver indeholder en af Amigaens komponenter også: ind-/udlæsningskomponenten. I en af disse chips løber hele tiden en 24 bits bred tæller, med en frekvens på 50 Hertz (i amerikanske apparater med 60 Hertz).

Disse bits kan dog ikke udlæses på een gang, f.eks. med en MOVE.L-ordre, da registeret er fordelt på 3 bytes. Den laveste af de tre bytes ligger på adressen $BFE801, den midterste på $BFE901 og den højeste med bits 16-23 på $BFEA01.

Et eksempel på anvendelsen af dette register: fastsættelsen af en underrutines kørselstid:

```
        ...
        bsr     gettime        ;øjeblikkelige tid til D7
        move.l  d7,d6          ;og gem den i D6
        bsr     rutine         ;start den rutine, der skal testes
        bsr     gettime        ;hent tiden igen
        sub.l   d6,d7          ;forløbne tid ligger med 1/50 sekund
        ...                    ;i D7

        ...
gettime:
        move.b  $bfea01,d7     ;HI-byte til D0
        lsl.l   #4,d7          ;forskydes to gange med 4,
        lsl.l   #4,d7          ;altså med 8 bits
        move.b  $bfe901,d7     ;tag tillige MID-byte'n
        lsl.l   #4,d7
        lsl.l   #4,d7          ;forskyd videre
        move.b  $bfe801,d7     ;og tillige LO-byte'n
        rts                    ;færdig
```

## 5.3. Hvordan man udlæser mus og joystick

Der findes to hardware-registre til musen og joystick'et, som viser disse indlæsningsapparaters tilstand, resp. position. Det er interessant, at en og samme port virker både med mus og joystick, selv om funktionen af de to er helt forskellig.

Joy-sticket har 4 kontakter, der sluttes ved en bevægelse og lægger massespændingen (-) på den, eller de, tilsvarende indgange i joystick-/mus-porten. Musen giver derimod mange og hurtige signaler, når den bevæges, to for horisontale og to for vertikale bevægelser.

Computeren må altså stadig overvåge de to porte, og ved en bevægelse bruge signalerne til at beregne musens nye position. Det er heldigvis ikke processorens opgave, da den ville få for meget at lave.

Vi kan finde mus-/joystick-portens status på adresserne $DFF00A for den forreste port (1) og $DFF00C for den bageste port (2). Disse ord indeholder informationer om den vertikale bevægelse af musen i den laveste byte, og om den horisontale i den højeste byte.

*Men pas på: adresserne må ikke udlæses med SEKA! Det fører nemlig af uudgrundelige årsager til at computeren crasher, hvad der ganske vist ser interessant ud (billedet begynder at danse), men kun lader sig ophæve med reset, og dermed totalt tab af alle data.*

For at kunne udlæse registrene, skal vi bruge et lille program:

```
joy = $dff00a

run:
      move   joy,d6       ;data 1 til d6
      move   joy+2,d7     ;data 2 til D7
      rts
```

Hvis man assembler programmet og starter det med "j run", får man indholdet af de to registre i D6 og D7. Bevæg nu musen en smule, og start programmet igen.

Som det ses, har værdien i D6 forandret sig. Skub nu musen horisontalt, så ændrer værdien i den laveste byte sig også, ved vertikal bevægelse værdien i den højeste byte.

Hvad vi har opnået, er ikke musepointerens absolutte position på skærmen. Det kan man fastslå, ved at skubbe musen helt op til venstre, udlæse værdien med "j run", og skubbe musen endnu længere til venstre. Musepointeren bevæger sig selvfølgelig ikke længere, men værdien i registret ændres alligevel. Som det ses, skal indholdet af registeret altid betragtes som relativt.

Lad os ændre programmet på følgende måde:

```
joy = $dff00a

run:
      move   d7,d6        ;den gamle position til D6
      move   joy,d7       ;den nye position til D7
      sub    d7,d6        ;differensen til D6
      rts
```

Start programet to gange. Resultatet i D6 er 0. Hvis musen nu bevæges, og der startes igen, får man, efter hver start, forskellen mellem den gamle og den nye position fra registret i D6. Dvs. den nuværende musepositions vertikale og horisontale afstand til den i sidste forsøg. Således kan man konstatere den relative bevægelse af musen mellem to udlæsninger.

Men nu til aflæsningen af joystick'et. Et joystick tilsluttes port to, og i programmet skal adressen $DFF00A ændres til $DFF00C. Når man atter starter programmet to gange, får man igen et nul i D6.

Bevæges joystick'et nu opad, og startes programmet, så giver det værdien $FF00. Der blev altså trukket 1 fra den højeste byte. Hvis man nu slipper joystick'et og starter igen, så finder man værdien $100 i D6: der blev adderet

1. Den samme effekt har en bevægelse af joystick'et mod venstre, efter at det slippes, subtraheres der atter 1.

De enkelte bevægelser og deres reultater med ovenstående program er:

| | | |
|---|---|---|
| op | **$FF00,** | **HI-byte -1** |
| ned | **$FFFF,** | **LO-byte -1** |
| til venstre | **$0100,** | **HI-byte +1** |
| til højre | **$0001,** | **LO-byte +1** |

Man skal ikke stole alt for meget på disse værdier. Hvis man rigtigt "rører rundt" med joysticket og kigger på resultatet, så finder man en vanvittig værdi i D6. Det skyldes, at hardwaren går ud fra, at der er tilsluttet en mus. Alligevel er metoden den hurtigste til at udlæse et joystick. Således kan også tilsluttes et eksternt apparatur, der giver anvendelige TTL-signaler til porten, som kan overvåges af et maskinprogram.

*Hvad vi mangler, for at kunne anvende joystick'et, er aflæsningen af affyringsknappen. Denne knaps tilstand indeholdes i bit 7 af den byte, som findes i lagercelle $BFE001. Hvis denne bit er sat, så er knappen ikke trykket ned. Det gælder for et joystick, der er tilsluttet port 2. Bit 6 af byte'n viser knappens tilstand, hvis joystick'et sidder i port 1, eller tilstanden af den venstre knap på musen.*

Vi bliver ved port 2. Hvis man vil teste den 7. bit af byte'n, for at udløse en eller anden funktion, hvis knappen er nedtrykket, så er det yderst simpelt. Bit 7 angiver jo en byte's fortegn. Det kan vi drage fordel af med følgende programstykke:

```
tst.b   $bfe001    ;affyringsknap 2 nedtrykket
bpl     fyr        ;ja, forgren
```

TST.B-ordren tester den adresserede byte og sætter i givet fald Z- og N-flagene. Hvis N-flaget er sat, så ved vi, at bit 7 i den testede byte er sat. Da affyringsknappen giver LO-potential, bliver bit'en slettet, hvis knappen er trykket ned og N-flaget tillige af TST-ordren. BPL-ordren i ovenstående program forgrener således, hvis der er trykket på knappen, da PL står for plus, og er opfyldt ved slettet fortegnsbit.

## 5.4. Betjening af parallel-interface't

En anden interessant mulighed for at koble eksterne apparater til Amigaen, er parallel-porten på computerens bagside, hvor man normalt tilslutter en printer. Dette stik giver flere dataledninger, hvormed man vilkårligt kan indlæse og udlæse data.

Internt ligger ved dette interface en byggesten, der kaldes PIA (Parallel Interface Adapter). Komponenten er udviklet specielt til parallel dataoverførelse. Der er ført 8 data- og nogle styreledninger fra chip'en til interface't.

Dataledningerne bruges både som udgange (som til printeren), og som indgange. Herunder kan man, bestemme retningen enkeltvis for hver af de 8 ledninger. F.eks. kan man definere tre indgangs- og fem udgangsledninger, og både overvåge og styre en ekstern proces.

Før vi ser nærmere på forbindelsen til Amigaens omverden, skal det nævnes, at tilslutningerne til Amiga-parallelporten arbejder med TTL-niveau. Dvs., der ind- og udlæses spændingsværdier på 0 volt for LO og 5 volt for HI. Når man laver en tilkobling til denne port, må man ikke overskride nævnte spænding. Desuden kan man ikke skabe nogen stor effekt med udgangssignalerne, så bør altid anvendes en driver. I modsat fald kan man ødelægge PIA-chip'en!

Lad os gå ud fra, at der er tilsluttet en TTL-kobling, som har ind- og udgange, til Amigaen. Som eksempel tager vi en ganske simpel kobling:

— en kontakt, der kan sætte portens bit 0 på LO-niveau og

— en lampe eller en lysdiode, der via en driver kan kobles til portens bit 1.

Lad kontakten stå i den stilling, hvor den er åben, og således ikke sætter noget niveau til porten. Sålænge porten nemlig står på udgang, må der ikke komme noget direkte signal, da HI ellers kunne blive til LO, og der således kunne opstå en kortslutning i PIA. Men lampen med driveren kan man roligt tilslutte, den vil så lyse.

I standardindstilligen er de 8 dataledninger stillet på indgang, da der kunne komme signaler. De data, der ankommer til porten, kan man finde i hardware-registret, der udgør PIA's dataregister. Men hvis man alllerede har benyttet printeren, så er retningen allerede omstillet til udgang. Så ændrer man indholdet af lagercelle $BFE301 til 0 med SEKA's "m"-ordre (forklaring følger).

Dataregistret har adressen $BFE101 og er 8 bits bredt, altså en byte. Adressen skal også programmeres med en bytetilgang, da det drejer sig om en ulige adresse.

Nu indtastes "m $bfe101" i SEKA. Her finder man værdien $FF, hvad der betyder, at alle portens indgange er åbne, og logisk nok er på HI. Oså bit 1, hvor lampen er tilkoblet, er på 1, fordi TTL-tilgangen altid er på dette niveau.

Når man indtaster en anden værdi og trykker return, venter man måske, at indholdet af den pågældende lagercelle også forandres. Men som det ses på udlæsningen, har værdien $FF ikke ændret sig. Lampen er heller ikke gået ud.

*Årsagen er, at ledningerne stadig står på indgang, og hardware-registret stadig indeholder den værdi, der svarer til de tilsluttede ledningers niveau. Man skal derfor trykke på escape-tasten, for igen at komme tilbage til SEKAs kommandofunktionsmåde.*

Nu lægges med kontakten et LO-signal på bit 0. Når man igen ser på indholdet af dataregistret med "q $bfe101", så finder man værdien $FE, som henviser til en slettet bit 0.

For at kunne tænde og slukke for lampen, må man omstille portens bit 1 til retningen "udgang". Informationen om, hvilke bits der er stillet på indgang og hvilke på udgang, indeholdes i PIAs dataretningsregister. Det pågældende register ligger på lageradresse $BFE301. Dette register har vi lige stillet på nul, for at sætte ledningerne på indgang.

Hver bit i byten svarer til en af de 8 dataledninger, bit 0 til ledning 0, bit 1 til ledning 1 osv. Hvis den tilsvarende bit slettes, omstilles ledningen til indgang, sættes den, får vi en udgangsledning.

Det vil vi afprøve på vort eksempel. Vi lader alle bits være på indgang, med undtagelse af bit 1, fordi lampen jo er tilsluttet ledning 1. Vi skal altså sætte bit 1 i dataretningsregistret på 1. For at opnå det, skal indtastes "m $bfe301", hvorpå SEKA angiver registerets øjeblikkelige indhold, et nul. Nu indtastes 2, og trykkes return. Som man kan se på udlæsningen, bliver værdien også skrevet i registret.

Nu kan man også slukke for lampen. Skriv hertil værdien 0 ind i dataregister $BFE101: lampen går ud.

Hvis man nu atter udlæser dette dataregister, så finder man værdien $FD. Man kan altså genfinde bit 1's tilstand, hvad der ofte er meget praktisk, når man skal fastslå, hvad der sidst blev skrevet i registret.

Nu vil vi skrive et program, som overfører kontaktens stilling direkte til lampen. Hvis kontakten er afbrudt, skal lampen lyse, ellers ikke.

For at afslutte programmet trykker man på affyringsknappen på et joystick, der er tilsluttet port 2. Programmet hertil er som følger:

```
datareg   = $BFE101       ;dataregister
dretnreg  = $BFE301       ;dataretnings-register
fyr       = $BFE001       ;affyringsknap til bit 7

run:
     move.b  #2,dretnreg  ;bit 1 på udgang
loop:
     move.b  datareg,d0   ;indgangsdata i D0
     lsl     d0           ;forskyd til venstre
     move.b  d0,datareg   ;og udlæs igen
     tst.b   fyr          ;test for affyringsknap
     bmi     loop         ;ikke trykket på: videre
     rts                  ;ellers slut
```

Nu kan kotakten sluttes og afbrydes, lampen vil vise tilstanden. Det blev opnået med LSL-ordren, der forskyder bits'ene fra D0 til venstre. Herved bliver bit 0 skubbet til bit 1, hvad der er nøjagtigt det, vi ville. Resultatet i D0 kan nemlig igen skrives direkte i dataregistret, og angiver kontaktens tilstand i bit 1.

En lille opgave: skriv programmet om, således at lampen går ud, når kontakten giver LO-potential.

8 bits er ikke nok til at betjene printeren. Der kræves ekstra signalledninger, der viser, at der står en gyldig byte på de 8 dataledninger, eller at printeren er parat til at modtage data.

Man kan finde ledningerne i registret på adresse $BFD0FE. Bits 0 til 2 er tilsluttet printerporten direkte.

**Bit 0 til BUSY**
**Bit 1 til POUT**
**Bit 2 til SEL i printerporten.**

Ledningerne anvendes altså, som ekstra indgang til at tilslutte eksternt apparatur.

## 5.5. Tonefrembringelse

Et særligt interessant tema for maskinsprogsprogrammører, er frembringelsen af toner og lyde. Amigaen giver mulighed for at afspille toner, lyde og/eller musikstykker som baggrund, ved hjælp af audio.devices og diverse I/O-strukturer. Metoden vil vi overlade til C- eller BASIC-programmørerne, idet vi direkte kan programmere audio-hardwaren med et lille maskinprogram!

Den i Amigaen inbyggede chip, der kaldes Paula, overtager meget af arbejdet, da den besidder alle kvalifikationer til tonefrembringelse. Da man også kan nå denne chip via hardware-registre for processoren, kan vi udnytte den direkte. Der er ikke noget bibliotek, i et højere programmeringssprog, der kan gøre mere end vi: programmere chip'en.

Hvordan det fungerer? Jo, da chip'en råder over sin egen lagertilgang (DMA = Direct Memory Access), behøver vi kun at sige til den, hvor i lageret vi har lagt dataene for den ønskede tone eler tonerække. Desuden må vi meddele, hvordan den skal fortolke disse data.

Til at begynde med det simpleste tilfælde: frembringelsen af en konstant tone. En sådan tone består af en svingning, der gentages hele tiden. Hvis vi laver en graf over svingningen, kan vi se dens kurveform. Der findes visse standardudgaver heraf: sinusform, rektangulær, trekantet eller savtak. Den simpleste af dem er den rektangulære svingning.

For at frembringe en tone med en rektangulær kurve, behøver vi kun at koble højttaleren til og fra. Den frekvens, det sker med, er også den hørbare tones frekvens.

Vi vil nu lokke sådan en tone ud af Amigaen. Dertil anlægger vi en tabel, hvor amplituden af den svingning, der skal frembringes, ligger. Til den rektangulære svingning behøver vi kun to tal i denne tabel: en stor og en lille værdi. Da sound-chip'en i Amigaen kender amplituder fra -128 til +127, ser tabellen f.eks. sådan ud:

```
soundtab:
   dc.b -100,100
```

Adressen på tabellen videregiver vi til sound-chip'en. Her har vi 4 muligheder, idet Amigaen har 4 sound-kanaler. Adressen på det hardware-register, hvori tabeladresserne til kanal 0 skal skrives, er $DFF0A0, til kanal 1 er den $DFF0B0, til kanal 2 $DFF0C0 og til kanal 3 $DFF0D0. Ved anvendelse af stereo-udgange gælder, at kanal 0 og 3 styrer den venstre højttaler, kanal 1 og 2 den højre. Vi vælger kanal 0 til vort eksempel og skriver derfor:

```
move.l  #soundtab,$DFF0A0  ;tabellens adresse
```

Det næste er, at vi skal meddele sound-chip'en, hvor mange data, der findes i tabellen. Disse data bliver hele tiden læst fra begyndelse til slutning, og videregivet til højttaleren. Hvis den er nået til slutningen, så begynder den forfra osv. Da sound-chip'en selv henter disse data fra lageret, og det altid sker i et ords bredde, men dataene selv er bytes, må tabellen hele tiden indeholde et lige antal bytes. Den længde, som skal videregives, angiver således også antallet af ord, dvs. antal bytes/2.

Længden overgives for kanal 0 til registeret på adressen $DFF0A4 (for kanal x adderes ganske enkelt x*$10!):

```
move    #1$dff0a4              ;tabellens længde i ord
```

Så følger angivelsen af, hvor hurtigt dataene skal udlæses og videregives til højttaleren. Dette ord angiver altså frekvensen, ganske vist på omvendt måde; jo højere værdi, desto lavere frekvens. Vi vælger værdien 600:

```
move    #600,$dff0a6           ;udlæsningshastighed
```

Til slut kan vi også bestemme lydstyrken, hvormed tonen eller lyden skal høres. Her kan vi vælge mellem 65 forskellige lydstyrker. Vi tager som mellemstærk lydstyrke værdien 40:

```
move    #40,$dff0a8            ;lydstyrke
```

Det er de data, som sound-chip'en behøver for at kunne frembringe tonen. Men hidtil er der ikke sket noget. Chip'en kan jo ikke vide, om de data, der står i registeret, er gyldige eller overhovedet skal give en lyd fra sig.

Lad os starte toneudlæsningen, resp. DMA'en for at vi også kan få noget at høre. Hertil tager vi DMA-kontrolregisteret på adressen $DFF096. Vi behøver kun 6 bits af dette ord.

| | |
|---|---|
| **Bit 15 ($8000)** | Når bit'en er sat, så bliver alle bits, der skrives ind heri, også sat i det interne register, ellers bliver de slettet. 0-bits påvirkes ikke. Det er vældig godt, idet ordet også indeholder DMA-informationer til diskettestyringen, som heller ikke skal forandres. |
| **Bit 9 ($200)** | Først denne bit muliggør DMA-lagerets tilgang til chip'en. Hvis vi vil starte tonen, må vi altså sætte bit 9. |
| **Bit 0-3** | Indkobler kanal 0-3, når bit'en er sat. |

Vi starter altså tonen ved at sætte bits 15,9 og 0:

```
move    #$8000+$200+1,$dff096  ;start DMA
```

Her er tonefrembringelsen på n gang. Denne gang med en sinuslignende tone:

```
;** Soundgenerering via hardware-register **

ctlw        = $dff096       ;DMA-control

c0thi       = $dff0a0       ;tabeladresse HI
c0tlo       = c0thi+2       ;tabeladresse LO
c0tl        = c0thi+4       ;tabellængde
c0per       = c0thi+6       ;afspilningshastighed
c0vol       = c0thi+8       ;lydstyrke
run:                        ;*frembring en enkel tone
    move.l  #table, c0thi   ;tabelbegyndelse
    move    #8,c0tl         ;tabellængde 8 ord
    move    #400,c0per      ;afspilningshastighed
    move    #40,c0vol       ;lydstyrke

    move    #8201,ctlw      ;start DMA/sound
    rts

table:                      ;sound-tabel: sinus
dc.b -40,-70,-90,-100,-90,-70,-40,0
dc.b 40,70,90,100,90,70,40,0
```

For at slå denne tone fra, behøver vi kun at sætte bit 0 af DMA-kontrolregistret tilbage. Til gengæld kan vi med et 0 i bit 15 indstille, at alle andre satte bits i det ord, der er skrevet i registret, slettes i chip'en selv. For at slette bit 0 skriver vi blot et 1 derind: bit 15=0 => bit 0 slettes. Her er den lille rutine, der stopper tonen fra kanal 0:

```
stille:                     ;* sluk for tonen
    move    #1,ctlw         ;sluk for kanal 1
    rts
```

Endnu en anvendelse af denne rutine: et program til frembringelse af en kort pippelyd, som man f.eks. kan anvende til tastaturklik.

```
;** Frembringelse af pippelyd **

ctlw        =$dff096        ;DMA-control

c0thi       = $dff0a0       ;Tabeladresse HI
c0tlo       = c0thi+2       ;tabeladresse LO
c0tl        = c0thi+4       ;tabellængde
c0per       = c0thi+6       ;afspilningshastighed
c0vol       = c0thi+8       ;lydstyrke
```

```
beep:                             ;*frembring kort pippelyd
        move.l #table,c0thi       ;tabelbegyndelse
        move   #4,c0tl            ;tabellængde
        move   #300,c0per         ;afspilningshastighed
        move   #40,c0vol          ;lydstyrke

        move   #$8201,ctlw        ;start DMA resp. lyd

        move.l #20000,d0          ;forsinkelsestæller
loop:
        dbra   d0,loop            ;tæl ned

still:
        move   #1,ctlw            ;og sluk for lyd
        rts

table:                            ;sound-tabel
        dc.b -40,-70,-40,0,40,70,40,0
```

Således kan man lade en til fire toner lyde på samme tid, uafhængigt af hinanden. Amigaen tilbyder en mulighed for at lade disse klange lyde mere spændende, vi kan modulere tonerne.
Antag, at man vil frembringe en sirenelyd. Det kan selvfølgelig opnås ved, at man programmerer hele hyletonesekvensen. Men det vil være et kæmpearbejde.
Det er lettere, hvis man anvender to tonekanaler. Lad os for eksempel tage kanal 1 til grundtonen og kanal 0 til dens modulation. Til en sirene må kanal 1, så indeholde hyletonens "envelope", og angive tonens stigen og falden med den rigtige hastighed.
Man har tre muligheder for at overlejre kanal 1 på kanal 0. Man kan styre lydstyrken fra kanal 0, dens afspilningshastighed resp. frekvens eller begge dele. Til vort eksempel behøver vi altså frekvensmodulationen.
Nu ændrer vi ovenstående program som følger:

```
:** Moduleret sound-generering via hardware-register **

ctlw   = $dff096          ;DMA-control
adkon  = $dff09e          ;audio/disk-control

c0thi  = $dff0a0          ;tabeladresse HI
c0tlo  = c0thi+2          ;tabeladresse LO
c0tl   = c0thi+4          ;tabellængde
c0per  = cothi+6          ;afspilningshastighed
c0vol  = c0thi+8          ;lydstyrke
```

```
run:
    move.l #table,c0thi+16 ;tabelbegyndelse kanal 1
    move   #8,c0tl+16      ;tabellængde
    move   #300,c0per      ;afspilningshastighed
    move   #40,c0vol+16    ;lydstyrke

    movel.l #table2,c0thi  ;tabelbegyndelse kanal 0
    move   #8, c0tl        ;tabellængde
    move   #60000,c0per    ;afspilningshastighed
    move   #30,c0vol       ;lydstyrke

    move   #$8010,adkon    ;modulationsmåde: FM
    move   #$8203,ctlw     ;start DMA
    rts

still:                     ;* sluk for lyd
    move   #$10,adkon      ;ingen modulation mere
    move   #3,ctlw         ;sluk for kanaler
    rts

table:                     ;data til grundtone
dc.b -40,-70,-90,-100,-90,-70,-40,0
dc.b 40,70,90,100,90,70,40,0

Table2:                    ;data til modulation
dc.w 400,430,470,500,530,500,470,430
```

Når man starter programmet, hører man en sirenelyd, som man kan forandre, som man vil.

Som det ses, er der dukket et nyt register op: "adkon". Registeret kontrollerer både modulationen af audiokanalerne og diskettefunktionen. Da der er tale om den samme teknik som ved DMA-kontrolregistret, hvor kun de specificerede bits sættes eller slettes (også her afhængigt af bit 15), behøver vi ikke at bekymre os om diskette-bits'ene. Men jeg råder til ikke at ekseperimentere med dem!

Foruden kontrolbit 15 er tillige bits 0-7 interessante. I disse bits bestemmes nemlig, hvilken audio-kanal, der modulerer hvilken anden kanal. Her gælder en vis begrænsning:

En kanal kan kun modulere den nærmeste højere kanal. Det er derfor, vi i det ovenstående eksempel valgte kanal 1 til grundtonen og kanal 0 til modulationen. Den modsatte vej er det lige så lidt muligt, som f.eks. at modulere kanal 3 med kanal 0. Derfor kan man heller ikke bruge kanal 3 til at modulere andre kanaler.

Her er en oversigt over bits 0-7 i "adkon"-registret med deres funktioner.

| Bit | Funktion |
|---|---|
| 0 | **Kanal 0 modulerer lydstyrken af kanal 1** |
| 1 | **Kanal 1 modulerer lydstyrken af kanal 2** |
| 2 | **Kanal 2 modulerer lydstyrken af kanal 3** |
| 3 | **Kanal 3 afbrydes** |
| 4 | **Kanal 0 modulerer frekvensen af kanal 1** |
| 5 | **Kanal 1 modulerer frekvensen af kanal 2** |
| 6 | **Kanal 2 modulerer frekvensen af kanal 3** |
| 7 | **Kanal 3 afbrydes** |

I eksemplet har vi sat bit 4, og således bestemt kanal 0 til at modulere frekvensen på kanal 1.

Når en kanal defineres som modulationskanal, ændrer betydningen af visse af dens parametre sig. Således er det unødvendigt at angive kanalens lydstyrke. Desuden skal tabellens data ikke længere betragtes som bytes, men som ord. Disse ord bliver så indført med den bestemte hastighed i det fastsatte register for den modulerede kanal (her afspilningsregistret).

Vil man ikke alene modulere frekvensen, men også lydstyrken af den anden kanal (her sætte bits 0 og 4 af "adkon"), bliver dataene igen fortolket lidt anderledes. Det første ord i tabellen bliver fortolket som lydstyrke, det andet som afspilningshastighed og så videre, alså stadigt skiftende. Dermed kan man f.eks. frembringe en forbikørende politibils sirenelyd.

## 5.6. Oversigt over hardware-registrene

De følgende tabeller skal give et overblik over de vigtigste hardware-registre. Beskrivelsen af hvert enkelt register ville sprænge rammerne af denne bog, så jeg må henvise til den relevante litteratur herom. Hvis man vil eksperimentere med disse registre, skal man tænke på, at det eventuelt kan føre, til at computeren crasher. Derfor bør man på forhånd sikre alle sine data på diskette og for en sikkerheds skyld tage den ud af stationen, som måske også kunne udføre fejlagtige funktioner.

Lad os begynde med de allerede omtalte PIA'er. Det drejer sig her om PIA-typen 8520, som der findes datablade om. Sådanne datablade kan jeg kun anbefale, da de viser alle komponentens muligheder. Man skal bare betænke, at visse funktioner og tilslutninger til 8520'eren allerede er fast integreret i Amigaen, og således betyder begrænsninger i anvendelsen af PIA'erne.

| PIA A | PIA B | Register-betydning |
|---|---|---|
| BFE001 | BFE000 | dataregister A |
| BFE101 | BFE100 | dataregister B |
| BFE201 | BFE200 | dataretningsregister A |
| BFE301 | BFE300 | dataretningsregister B |
| BFE401 | BFE400 | taktgiver A LO |
| BFE501 | BFE500 | taktgiver A HI |
| BFE601 | BFE600 | taktgiver B LO |
| BFE701 | BFE700 | taktgiver B HI |
| BFE801 | BFE800 | begivenhedsregister bits 0-7 |
| BFE901 | BFE900 | begivenhedsregister bits 8-15 |
| BFEA01 | BFEA00 | begivenhedsregister bits 16-23 |
| BFEB01 | BFEB00 | ubenyttet |
| BFEC01 | BFEC00 | serielt dataregister |
| BFED01 | BFED00 | interrupt-kontrolregister |
| BFEE01 | BFEE00 | kontrolregister A |
| BFEF01 | BFEF00 | kontrolregister B |

Nogle interne betydninger:

**$BFE101** **Dataregister for parallel-interface't.**

**$BFE301** **Dataretningsregister for parallel-interface't.**

**$BFEC01** **Tastaturets tilstand, indeholder den sidst trykkede specialtast (Shift, Alternate, Control, Amiga).**

Nu følger de registre, hvorunder tonefrembringelsen sorterer. Her skal man specielt behandle de to første registre med forsigtighed, da de kan udløse ubehagelige effekter ved fejlbehandling.

Registrene kan enten kun læses eller skrives i. Det angives af R/W- (Read/Write-) oplysningerne i listen.

| Adresse | R/W | Betydning |
|---|---|---|
| DFF096 | W | skriv DMA-kontrol |
| DFF002 | R | læs status af DMA-kontrol og blitter |
| -- Audiokanal 0 -- | | |
| DFF0AA | W | dataregister |
| DFF0A0 | W | pointer på tabelbegyndelse bits 16-18 |
| DFF0A2 | W | pointer på tabelbegyndelse bits 0-15 |
| DFF0A4 | W | tabellængde |
| DFF0A6 | W | afspilningshastighed/periode |
| DFF0A8 | W | lydstyrke |

| Adresse | R/W | Betydning |
|---|---|---|
| -- Audiokanal 1 -- | | |
| DFF0BA | W | dataregister |
| DFF0B0 | W | pointer på tabelbegyndelse bits 16-18 |
| DFF0B2 | W | pointer på tabelbegyndelse bits 0-15 |
| DFF0B4 | W | tabellængde |
| DFFOB6 | W | afspilningshastighed/periode |
| DFF0B8 | W | lydstyrke |
| -- Audiokanal 2 -- | | |
| DFF0CA | W | dataregister |
| DFF0C0 | W | pointer på tabelbegyndelse bits 16-18 |
| DFF0C2 | W | pointer på tabelbegyndelse bits 0-15 |
| DFF0C4 | W | tabellængde |
| DFF0C6 | W | afspilningshastighed/periode |
| DFF0C8 | W | lydstyrke |
| -- Audiokanal 3 -- | | |
| DFF0DA | W | dataregister |
| DFF0D0 | W | pointer på tabelbegyndelse bits 16-18 |
| DFF0D2 | W | pointer på tabelbegyndelse bits 0-15 |
| DFF0D4 | W | tabellængde |
| DFF0D6 | W | afspilningshastighed/periode |
| DFF0D8 | W | lydstyrke |

Nu til de registre, hvor informationerne om joystick, mus eller potentiometre indeholdes. Disse adresser er allerede tildels dukket op.

| Adresse | R/W | Betydning |
|---|---|---|
| DFF00A | R | joystick/mus port 1 |
| DFF00C | R | joystick/mus port 2 |
| DFF012 | R | potentiometerpar 1 tæller |
| DFF014 | R | potentiometerpar 2 tæller |
| DFF018 | R | potentiometer-tilslutninger indgang |
| DFF034 | W | potentiometer-port retning |

# 6. Hvordan man arbejder med Amiga-operativsystemet

Nu vil vi gøre endnu et skridt henimod en hensigtmæssig assembler-programmering, idet vi indvier Amigaen i vore ønsker. Det er ikke nok at have en tekst liggende i lageret, man vil også gerne kunne få den ud på skærmen. Men ved læseren, hvordan man får et bogstav ud på skærmen? For ikke at tale om, hvordan man tegner et vindue på skærmen, som endda lader sig ændre med musen?

Nuvel, vi behøver ikke at vide disse ting præcist. Heldigvis råder Amigaens operativsystem over rutiner, der kan klare den slags for os. Rutinerne er sammenfattet i såkaldte biblioteker, som vi vil kigge nærmere på.

## 6.1. Hvordan man load'er biblioteker

Før vi kan anvende et bibliotek, må det først stilles til rådighed, dvs. load'es i lageret. Man load'er altid hele biblioteket, også selv om man kun har brug for en af funktionerne.

Først må man overveje, hvad ens eget program skal kunne, altså hvilke biblioteker, man har brug for. Til simpel ind- og udlæsning af tekster behøver man ikke et bibliotek, der indeholder rutiner til bevægelsesgrafik.

På den normale workbench-diskette er der en hel del biblioteker. Her er en oversigt over deres navne og anvendelsesområder:

*exec.library*
Dette bibliotek har man brug for, til at load'e andre biblioteker. Det ligger derfor allerede i lageret, og behøver således ikke at load'es. Med det muliggøres grundlæggende funktioner som lagerpladsreservering eller bearbejdning af I/O-kanaler.

*dos.library*
Indeholder alle funktioner til normale ind-/udlæsningsoperationer som betjening af monitor eller diskette.

*intuition.library*
Bruges til bearbejdning af screens, vinduer, menuer osv.

*clist.library*
Indeholder rutiner til bearbejdning af copper-lists, hvormed billedopbygningen kan styres.

*console. library*
Indeholder grafikrutiner til tekstudlæsning i consol-vinduer.

*diskfont. library*
Kræves til anvendelse af tegnsæt, der er lagret på disketten.

*graphics. library*
Indeholder funktioner til styring af blitteren resp. grafik-chips, og er derfor nødvendig for de grafiske grundfunktioner.

*icon. library*
Anvendes til oprettelsen og bearbejdningen af workbench-symboler.

*layers. library*
Er beregnet på bearbejdelse af skærmbilledlageret (Layers).

*mathffp. library*
Indeholder de grundlæggende matematisk operationer til tal med flydende decimaltegn.

*mathieedoubbas. library*
Giver de matematiske grundfunktioner til hele tal (integers).

*mathtrans. library*
Indeholder højere funktioner som vinkel- eller logaritmeberegninger.

*potgo. library*
Tjener til udnyttelsen af analogt input i Amigaen.

*timer. library*
Giver rutiner til rådighed for tidskritiske programmer, hvormed der kan programmeres eksakte tidsintervaller.

*translator. library*
Indeholder som den eneste funktion "Translate", der omsætter en normal tekst til den fonetiske skrivemåde for "Narrator"en, sprogsynthesizeren.

Man kunne selvfølgelig altid åbne alle disse biblioteker, resp. load'e dem. Men man må betænke, at det for det første kræver tid og for det andet lagerplads. Af den grund bør man altid nøje overveje, hvilke funktioner, man har brug for, og i hvilke biblioteker de ligger.

Lad os antage, at vi ville skrive et program, der skal udføre tekstind-/udlæsning. Her har vi brug for biblioteket "dos library", som vi nu vil load'e.

Load'ningsprocessen er en opgave for "exec.library". Biblioteket indeholder bl.a. funktionen "OpenLib", der skal kaldes med angivelse af de fornødne parametre.

## 6.2. Hvordan man kalder funktioner

Ethvert library begynder i lageret med en række JMP-ordrer. Disse JMP'er ligger smukt på række, og forgrener sig til de rutiner, der indeholder biblioteket. For at kalde en funktion, må man altså først finde begyndelsen af denne JMP-tabel, og derfra kalde funtionen x med et hop til den x'te JMP. Almindeligvis sker det ved hjælp af et offset, hvormed en JSR starter funktionen. Da man normalt ikke får begyndelsen, men afslutningen af JMP-tabellen som basisadresse, er disse offsets negative.

I praksis ser det ret simpelt ud. For som i eksemplet ovenfor at load'e "dos library" behøver vi altså først "exec-library"s basisadresse. Det er adressen $000004. Ved kald af en funktion fra et andet bibliotek, skal naturligvis anvendes en anden basisadresse, som man også får, når man loader library'et.

Nu behøver vi offset'et for den ønskede funktion. Det er funktionen OpenLib, der har offset'et -408. Alle funktions-offsets findes i appendikset.

Til OpenLib-funktionen behøver vi en pointer på navnet på det bibliotek, der skal load'es, her "dos.library", samt et langt lagerord, hvor vi kan lagre DOS-biblioteks basisadresse, som OpenLib-funktionen returnerer. Det er vigtigt at angive biblioteksnavnet med *små* bogstaver, da åbningen ellers ikke fungerer.

Og således ser alt det ud, som program:

```
;** Load'ning af DOS-biblioteket "dos.library" **

execbase = 4                    ;basisadresse på EXEC-biblioteket
OpenLib  = -408                 ;offset for OpenLib-funktionen

start
        move.l  execbase,a6     ;basisadresse til A6
        lea     dosname,a1      ;adresse på biblioteksnavn
        moveq   #0,d0           ;versionsnummer
        jsr     Openlib(a6)     ;kald af funktionen
        move.l  d0,dosbase      ;gem DOS-basisadresse
        beq     error           ;hvis nul, så fejl!
```

```
    ...
    ...                         ;fortsat program...

error:
    ...                         ;fejl optrådt
    ...

dosname                         ;navn på det bibliotek der skal åbnes
    dc.b    'dos.library',0,0, even
dosbase:                        ;plads til DOS-basisadressen
    blk.l   1
```

Sådan bliver DOS-biblioteket load'et og stillet til rådighed. Alle andre biblioteksfunktioner kaldes på den måde, hvorunder parametrene bliver overgivet til forskellige registre. For det meste er dataregistret D0 nul i tilfælde af fejl; altså hvis funktionen ikke blev udført korrekt.

Når programmet har gjort sit arbejde, så skal alle åbnede biblioteker lukkes, før man vender tilbage til workbench'en eller CLI. Det overtages af funktionen CloseLib med offset'et -414. Den ligger ligesom OpenLib i EXEC-biblioteket, og kræver blot basisadressen på det bibliotek, der skal lukkes, som parameter. Således ser programdelen til at lukke "dos.library" ud, efter den foretagne åbning:

```
CloseLib = -414

    ...
    move.l  execbase,a6     ;EXEC-basisadresse
    move.l  dosbase,a1      ;DOS-basisadresse
    jsr     CloseLib(a6)    ;bibliotek lukkes
```

# 6.3. Programforberedelse

Før man lader et program begynde arbejdet, må man sædvanligvis træffe nogle forberedelser. Hermed menes initialiseringen, som programmet skal udføre.

Tag et program, der skal repræsentere en mindre tekstbehandling. Sådan et program skal kunne lagre tekster, hvilket kræver et lagerområde. Desuden skal der finde tastaturindlæsning og udlæsning på skærmen sted, hvad der kræver et egnet udlæsningsvindue.

For at opfylde disse krav, må der først åbnes et eller flere biblioteker. Lad os forudsætte, at DOS-biblioteket er load'et, så vi kan gå i gang med næste trin.

## 6.3.1. Reservering af lagerplads

Til at få tildelt et lagerområde af operativsystemet, findes der flere muligheder resp. funktioner. En af dem må vi anvende, for at der ikke skal forekomme overlapning af de anvendte lagerområder, når flere programmer køres samtidig (multitasking).

Se på den funktion, der anvendes i normale tilfælde. Funktionen ligger i det interne EXEC-bibliotek og bærer navnet AllocMem (offset -$c6). Den reserverer et lagerområde af den ønskede længde (i D0), som ligger i lageret. Den virkelige adresse, hvorfra lageret reserveres, får man returneret i dataregister D0. Hvis returværdien er 0, så kunne den ønskede lagerstørrelse ikke reserveres.

Foruden lagerlængden overgives også et "modusord" til D1. I modusordet kan det bestemmes, om den reserverede lagerplads skal slettes direkte eller ej.

Og sådan ser hele programdelen ud:

```
execbase  = 4
AllocMem = -$c6

     ...
     move.l  #antal,d0      ;antal bytes der skal reserveres
     move    #modus,d1      ;modusord
     move.l  execbase,a6    ;DOS-basisadresse i A6
     jsr     AllocMem(a6)   ;funktionskald
     move.l  d0,adresse     ;gem lagerbegyndelse
     beq     fejl           ;lager ikke reserveret
     ...
```

Den anden mulighed for lagerreservering er en funktion ved navn AllocAbs (offset -$CC). Hermed bliver, modsat AllocMem-funktionen, reserveret et ganske bestemt lagerområde. Til den ende overgives antallet af bytes, der skal reserveres, til D0 og den ønskede startadresse til adresseregister A1. Også den funktion giver et nul tilbage i D0, hvis det pågældende lagerområde ikke kunne reserveres.

```
execbase = 4
AllocAbs = -$cc

     ...
     move.l  #antal,d0      ;antal bytes, der skal reserveres
     lea     adresse,a1     ;ønskede startadresse
     move.l  execbase,a6    ;EXEC-basisadresse
```

```
        jsr     AllocAbs(a6) ;reserver lager
        tst.l   d0           ;alt OK?
        beq     fejl         ;nej !
        ...
```

Når programmet har gjort sit arbejde, skal den reserverede lagerplads gives tilbage til systemet, før der vendes tilbage til CLI resp. workbench. Hertil tjener funktionen FreeMem (offset -$d2).

Til funktionen bliver byte-antallet i D0, samt startadressen på lageret, overgivet ligesom ved AllocAbs. Men her skal man lægge mærke til noget specielt:

Afmeldingen af et lagerområde, der ikke var reserveret, fører normalt til, at computeren crasher!

Og sådan ser befrielsen, af det besatte lagerområde, ud i programmet:

```
execbase = 4
FreeMem = -$d2

        ...
        move.l  #antal,d0     ;antal bytes, der skal afmeldes
        lea     adresse,a1    ;startadresse for AllocAbs
        move.l  execbase,a6   ;EXEC-basisadresse
        jsr     FreeMem(a6)   ;frigiv lager
        tst.l   d0            ;alt OK?
        beq     fejl          ;nej ! Men dog!
        ...
```

## 6.3.2. Åbning af et vindue på nemmeste måde

Den ejendommelige overskrift, over dette afsnit, vækker måske undren. Men forskellen mellem de to måder, at åbne et vindue på, er så stor, at begge ikke kan behandles i samme afsnit.

Den måde, der vises her, er meget simpel, men byder ikke på muligheden for at arbejde med vilkårlige "gadgets". Gadgets er f.eks. lukkesymbolet i det øverste venstre hjørne af mange vinduer, eller størrelsesændringssymbolet forneden til højre.

Hvis man åbner et vindue på den simple måde, er næsten alle gadgets til stede, men lukkesymbolet mangler dog. Det er grunden til, at metoden ikke egner sig til alle formål. Men nu til sagen:

Til at åbne et vindue bruges en funktion i DOS-biblioteket, der naturligvis må være åbnet (se kapitlet "Hvordan man load'er et bibliotek"). Det er den almindelige OPEN-funktion, som man kan bruge til en hel bunke ting. Det kan anbefales at indbygge en "open"-underrutine i et program, som man bruger flere gange.

Først de grundlæggende trin:

```
;** Load'ning af DOS-biblioteket "dos library" **

execbase = 4        ;basisadresse for EXEC-biblioteket
OpenLib  = -408     ;offset for OpenLib-funktionen
Open     = -30      ;offset for DOS-funktionen OPEN

start:
    move.l  execbase,a6     ;basisadresse i A6
    lea     dosnavn,a1      ;adresse på biblioteksnavn
    moveq   #0,d0           ;versionsnummer: ligegyldigt
    jsr     OpenLib(a6)     ;kald af funktionen
    move.l  d0,dosbase      ;gem DOS-basisadresse
    beq     error           ;hvis 0, så fejl !

    ...
    ...                     ;åben nu vindue osv.
    ...

error:
    ...                     ;fejl optrådt
    ...

do_open:                    ;almen OPEN-funktion
    move.l  dosbase,a6      ;DOS-basisadresse til A6
    jsr     Open(a6)        ;kald OPEN-funktion
    tst.l   d0              ;test om OK
    rts                     ;færdig, vurder test senere

dosname                     ;navn på biblioteket, der skal åbnes
    dc.b    'dos library',0,0, even
dosbase                     ;plads til DOS-bsisadresse
    blk.l   1
```

Rutinen bærer navnet "do_open", da label'en "Open" allerede er anvendt til offset'et. Denne rutine kalder blot Open-funktionen, der gemmer sig i DOS-biblioteket.

Funktionen skal også overgives parametre, der angiver, hvad der skal åbnes. Disse parametre overgives i registrene D1 og D2, således:

D1 peger på en definitionsblok for det, der skal åbnes. Det betyder, at der skal stå et filnavn, som afsluttes med en nul-byte. D1 skal, som alle adresser, overgives som et langt ord.

D2 indeholder den modus, hvormed funktionen skal udføres. Her skelnes mellem modus gammel (1005) og ny (1006). Et af disse tal skal hver gang overgives i et langt ord til D2.

Tilbage til opgaven at åbne et vindue. Heldigvis tillader Amiga-DOS, at anvende ind- og udlæsningskanaler på n og samme måde. Til disse standardkanaler hører foruden diskettefiler også konsollen (tastatur og monitor), printerinterface't og det serielle RS232-interface.

*Os interesserer nu konsol ind-/udlæsningen. Hvis vi nemlig, angiver konsollen som filnavn for den kanal, der skal åbnes, så åbnes automatisk et vindue.*

Angivelsen begynder navnet med CON: svarende til DF0: ved disketteoperationer. Der kræves dog flere oplysninger, om det vindue der skal åbnes.

Disse oplysninger er X- og Y-koordinater for det øverste venstre hjørne af vinduet, og for det nederste højre hjørne samt det navn, der skal stå på vinduets titellinie. En fuldstændig definitionsblok for vinduet ser ud som følger:

```
vinduesnavn: dc.b 'CON:0/10/640/200** Test-vindue **',0
```

For at åbne vinduet skal denne linie, samt de følgende programlinier, indføjes i det ovenstående program.

```
        ...
        move.l  #vinduesnavn,d1   ;pointer på definition
        move.l  #1005,d2          ;modus: gammel
        bsr     #do_open          ;åben vindue
        beq     error             ;gik ikke
        move.l  d0,handle         ;ellers gem kendingsnummer
        ...

handle:         blk.l 1           ;plads til kendingsnummer
```

Her er to punkter, der skal afklares:

Den overgivne modus skal ved åbningen af et vindue angives med "gammel". Da vinduet logisk set ikke eksisterede, før det blev åbnet, er det temmelig uforståeligt, men det gør jo ikke noget...

Den parameter, der blev returneret fra "do_open" rutinen i register D0 er nul i tilfælde af fejl, altså hvis åbningen ikke er lykkedes. Ellers repræsenterer værdien den åbnede kanals identifikationsnummer (her vinduets), det såkaldte handle-nummer, forkortet kaldet "handle".

*Nummeret skal opbevares omhyggeligt, da man altid skal angive det ved enhver funktion, der skal tilgå denne kanal. I eksemplet blev nummeret lagret i det lange ord "handle".*

Det åbnede vindue besidder ikke noget lukkesymbol, men det lader sig, efter ønske, forstørre eller formindske, forskyde eller lægge foran eller bagved andre vinduer. Disse manipulationer, som man kan gennemføre med musen, bliver bearbejdet fuldstændigt af Amigaen (i modsætning til ATARI ST, hvor et program også må bekymre sig om disse ting).

En vigtigt funktion som dette nummer kræver, er den, der atter lukker kanalen og i vort tilfælde dermed vinduet. Funktionen ligger ligesom Open-funktionen i DOS-biblioteket og har navnet "Close". Den har offset -36, og kræver kun en parameter: handle-nummeret på den kanal, der skal lukkes i register D1.

For at lukke vinduet, behøver man følgende linier i programmet:

```
Close = -36

    ...
    move.l  handle,d1      ;handle-nummer i D1
    move.l  dosbase,a6     ;DOS-basisadresse i A6
    jsr     Close(a6)      ;og luk kanal
```

Og væk er vort vindue!

Endnu en bemærkning om åbning og lukning af vinduer på den måde: hvis man åbner flere vinduer på samme måde, får man også flere vinduer, og dermed flere handle-numre. Således kan man afbilde og bearbejde et vilkårligt antal vinduer på skærmen, og selvfølgelig også lukke vilkårlige vinduer igen.

Endnu en enkel metode til at åbne vinduer. Man overgiver simpelthen navnet RAW: som kanalbetegnelse i stedet for CON:. Alle andre parametre og operationer forbliver de samme.

Afprøver man det en gang, så findes ingen synlig forskel på de to slags vinduer. De ser ens ud, og lader sig bearbejde med musen på samme måde. Forskellen bliver først synlig, når man indtaster noget i disse vinduer. Så bliver man klar over, at cursor-tasterne også fungerer, i modsætning til med CON:-vinduer som CLIs, hvor disse taster bliver ignoreret.

## 6.4. Ind- og udlæsning

Ved siden af administration og beregning af data er et programs vigtigste funktion, at ind- og udlæse disse data. Herunder hører alle former for dataoverførelse til eller fra lageret, som f.eks. skærmbilled- eller printerudlæsning, tastaturindlæsning, betjeningen af det serielle eller parallele interfaces, udsendelse af lyd eller sprog og ikke mindst disketteoperationerne.

Vi skal lære alle metoderne til dataind- og -udlæsning under programmering og anvendelse. Herunder vil blive udviklet nogle programmer, der p.g.a. deres opbygning som underrutiner, vil være nyttige i senere programmer. Det anbefales at anlægge et bibliotek med underrutiner, som enten kan integreres i et nyt program eller linkes til det.

Forudsætningen for en ind-/udlæsning er, at man har forberedt data til udlæsning eller plads til indlæsning. For at gøre det, må der først programmeres en korrekt programbegyndelse, hvor EXEC- og DOS-biblioteket åbnes, og der reserveres lagerplads. Når det er sket, begynder de fleste programmer med at udlæse en tekst, som kan være en programoverskrift, eller en opfordring til at indlæse data via tastaturet, et såkaldt "prompt" eller klartegn. Lad os begynde med teknikken ved skærmbilledudlæsninger.

### 6.4.1. Skærmbilledudlæsninger

Med Amiga dukker det første spørgsmål op her: hvorhen skal skærmbilledudlæsningen gå? Med andre computere er spørgsmålet trivielt. Der eksisterer kun en billedskærm, som udlæsningerne kan ses på. Med Amigaen må man straks bestemme, i hvilket vindue der skal skrives.

Der er to muligheder:

1. **Udlæsning i CLI-vinduet.**
2. **Udlæsning i et særligt vindue**

Den første mulighed er afhængig af, om man har startet det program, som skal foretage udlæsningerne, fra CLI. Er det ikke tilfældet, så skal vi åbne et særligt vindue. Hvis det er tilfældet, kan vi også bruge det vindue, som CLI allerede har åbnet, til vore udlæsninger.

Til den anden mulighed skal man, altså først åbne et vindue. Som vi har set, er der tre metoder at vælge imellem. Til en simpel udlæsning af tekst og tegn er forskellen mellem dissse tre vinduestyper ikke så betydende, så vi har endnu frit valg af hvilken type vindue, vi vil anvende. Jeg foreslår derfor, at vi åbner et CON:-vindue og lagrer dets handle-nummer i "conhandle".

Vi har altså åbnet vinduet og vil udlæse en overskrift. Vi vælger en tekst, og skriver en tilsvarende linie i programmet, hvormed teksten med sin nul-afslutning bringes ind i lageret. Sådan en linie ser omtrent således ud.:

```
titel: dc.b "** Velkommen til dette program ! **"
titelende:
even
```

"even" er en lille tilføjelse, der er meget hensigtmæssig, når der følger enten ord-data eller programlinier efter teksten. "even"-kommandoen får assembleren til, ifald der er behov for det, at indføje en ekstra nulbyte, for at den næste adresse også skal blive lige.

For nu at udprinte teksten, får vi brug for en DOS-funktion til: write. Den har offset -48 og kræver tre parametre:

**I D1** Handle't for den åbnede udlæsningskanal, hvori der skal skrives. I vort tilfælde er det handlenummeret på vinduet, som er modtaget fra Open-ordren.

**I D2** Adressen på den tekst, der skal udlæses, i vort eksempel adressen "titel", og

**I D3** Antallet af tegn, der skal udlæses, i bytes.

For at kende antallet af bytes, der skal udlæses, skulle vi tælle tegnene i teksten. Label'en "titelende" er tilføjet for at undgå optællingen. Med labelen kan assembleren selv udregne længden af teksten (hvad har man ellers en computer til?), når man skriver:

```
move.l    #titelende-titel,d3
```

Nødvendigheden af at angive tekstlængden har i øvrigt den fordel, at der også kan ligge styretegn mellem begyndelse og afslutning af teksten, hvormed visse forhold ved tekstudlæsning kan klares med et slag. Disse styretegn skal vi lære senere.

Her kommer hele programdelen:

```
write = -48
          ...                              ;åben vindue
          ...
          move.l  dosbase,a6               ;DOS-basisadresse
          move.l  conhandle,d1             ;overgiv handle
          move.l  #titel,d2                ;tekstadresse
          move.l  #titelende-titel,d3      ;og længde
          jsr     write(a6)                ;kald funktion
          ...
```

```
titel: dc.b "** Velkommen til dette program ! **"
titelende:
even
```

Funktionerne vil man ofte få brug for. Det forekommer tillige, at man kun vil udlæse et enkelt tegn. Til at dække disse to funtioner bruges derfor dette underprogram, der har fire indhoppunkter:

**pmsg** Udlæser teksten fra (D2) til en afsluttende nulbyte.

**pline** Som ovenfor, blot udføres automatisk et CR efter teksten, så cursoren sættes på begyndelsen af den næste linie.

**pchar** Udlæser tegnet i D0.

**pcrlf** Sætter cursoren på begyndelsen af næste linie.

Her er underprogrampakken:

```
write = -48

    ...
pline:                          ;* udlæs linie med CR
    bsr     pmsg                ;udlæs linie

pcrlf:                          ;* cursor til næste linie
    move    #10,d0              ;linefeed
    bsr     pchar               ;udlæs
    move    #13,d0              ;og CR

pchar:                          ;* udlæs tegn i D0
    move.b  d0,outline          ;tegn i udlæsningsbuffer
    move.l  #outline,d2         ;adresse på tegnet

pmsg:                           ;*udlæs linie fra (D2) til 0-byte
    move.l  d2,a0               ;adresse i A0
    clr     d3                  ;længde =0

ploop:
    tst.b   (a0)+               ;nulbyte?
    beq     pmsg2               ;ja: længden fundet
    addq.l  #1,d3               ;ellers længde+1
    bra     ploop               ;og søg videre
pmsg2:
    move.l  dosbase,a6          ;DOS-basisadresse i A6
```

```
        move.l  conhandle,d1    ;vort vindue-handle
        jsr     write(a6)       ;kald write-funktion
        rts                     ;færdig

outline:        dc.w 0          ;udlæsningsbufer til 'pchar'
conhandle:      dc.l 0          ;vinduets handle
```

Ved hjælp af programmet kan man bringe vilkårlige udlæsninger ind i CON:-vinduet. Disse rutiner virker også, når man har åbnet et RAW:-vindue, men så skal man også omdøbe label'en "conhandle" til "rawhandle", for at der ikke senere sker forvekslinger.

Lad os blive ved CON:-vinduerne. Der findes foruden de normale tegn tillige specielle tegn til udlæsning, som ikke bliver synlige, men udløser en funktion eller ændrer parametre for udlæsningen. Det kaldes styretegn.

Vi har allerede lært et af styretegnene at kende, nemlig "linefeed" ($A). Dette tegn bliver heller ikke udlæst, men udløser den funktion, der bringer cursoren frem til den næste linie, og i givet fald endda skubber vinduets indhold en linie op. Dette kan være meget nyttigt, men som det ses, findes også meget mere interessante styretegn.

Her følger en liste over styretegn, der udløser en funktion. Disse tegn er opført som hexadecimaltegn.

| Sekvens | Funktion |
|---|---|
| **08** | backspace |
| **0A** | linefeed, cursor ned |
| **0B** | cursor en linie op |
| **0C** | slet vindue |
| **0D** | carriage return, cursor til første spalte |
| **0E** | skift til normalrepræsentation (tag 0F tilbage) |
| **0F** | skift til specialtegn |
| **1B** | escape |

De følgende sekvenser begynder med tegnet $9B, det såkaldte CSI (Control Sequence Inducer). De tegn, som følger efter tegnet, bevirker så en funktion. De værdier, der er anført i firkantede parenteser, kan bortfalde. Angivelsen af "n" skal ske i ASCII-tegn, som et decimaltal med et eller flere cifre. Den værdi, der antages, hvis n udelades, er anført i parenteser bagved n.

| Sekvens | Funktion |
|---|---|
| 9B [n] 40 | indføj n tomme mellemrum |
| 9B [n] 41 | cursor n (1) linie(r) op |
| 9B [n] 42 | cursor n (1) linie(r) ned |
| 9B [n] 43 | cursor n (1) linier til højre |
| 9B [n] 44 | cursor n (1) linier til venstre |
| 9B [n] 45 | cursor n (1) linier ned i spalte 1 |
| 9B [n] 46 | cursor n (1) linier op i spalte 1 |
| 9B [n] [3B n] 48 | cursor på pågældende linie og spalte |
| 9B 4A | slet vindue fra cursor |
| 9B 4B | slet linie fra cursor |
| 9B 4C | indføj linie |
| 9B 4D | slet linie |
| 9B [n] 50 | slet n tegn fra cursor |
| 9B [n] 53 | skub n linier op |
| 9B [n] 54 | skub n linier ned |
| 9B 32 30 68 | med det samme: linefeed => linefeed + return |
| 9B 32 30 6C | med det samme: linefeed => kun linefeed |
| 9B 6E | send cursorposition! Der returneres en tegnstreng af følgende form:<br>9B (linie) 3B (spalte) 52 |
| 9B (stil);(forgrundsfarve);(baggrundsfarve) 6D | De tre parametre er ligeledes decimaltal i ASCII-format. De betyder:<br>Stil: 0 = normal<br>1 = fed skrift<br>3 = kursiv<br>4 = understreget<br>7 = invers<br>Forgrundsfarve: 30-37<br>Farve 0-7 for tekst<br>Baggrundsfarve 40-47<br>Farve 0-7 til baggrund |
| 9B (længde) 74 | fastsætter maksimale antal linier, der skal repræsenteres. |
| 9B (bredde) 75 | fastsætter maksimale linielængde |
| 9B (afstand) 78 | definerer afstanden i pixels fra vinduets venstre kant, hvor udlæsningen skal begynde |
| 9B (afstand) 79 | definerer afstanden i pixels fra vinduets øvre kant, hvorfra der skal udlæses<br>De sidste fire funktioner kan bringes til normalstillingen ved at udelade parameteren. |

| Sekvens | Funktion |
|---|---|
| **9B 30 20 70** | **gør cursor usynlig** |
| **9B 20 70** | **gør cursor synlig** |
| **9B 71** | **send vinduets mål! Der returneres en tegnstreng af følgende form:** |
| | **9B 31 3B 31 3B (linier) 3B (spalter) 73** |

For at demonstrere anvendelsen af disse styretegn, indtastes følgende tekst i vinduet med "pmsg"-rutinen:

```
text: dc.b $9b,"4;31;40m"
      dc.b "overskrift"
      dc.b $9b,"3;33;40m",$9b,"5;20H"
      dc.b "** Hallo, verden !**",0
```

Parametrene til styretegnene er markeret med anførselstegn og angivet som en ASCII-tegnstreng. Som det ses, kan man ret let opnå virkningsfulde tekstudlæsninger!

Efter vi har beskæftiget os med udlæsning af tegn og tekst, kommer vi til det modsatte, nemlig indlæsningen af tekst.

## 6.4.2. Indlæsning med tastaturet

Til indlæsning af tegn via tastaturet kan benyttes en enkel metode. Der læses fra CON:-vinduets allerede åbnede ind-/udlæsningskanal. Dertil kræves endnu en funktion fra DOS-biblioteket. Funktionen hedder READ og har OFFSET'et -42.

Ligesom til WRITE-funktionen overgives tre parametre til READ-funktionen, det er:

**I D1** Handle-nummeret, som kommer fra OPEN-funktionen.

**I D2** Den adresse, hvorfra de læste data skal lagres i lageret, og

**I D3** antallet af data, som skal læses.

Hertil kommer den sædvanlige underrutine, der indlæser så mange tegn fra tastaturet, resp. fra konsollen i en buffer ved navn "inline", som blev overgivet i D3.

```
Read    = -42

    ...
readchr:                        ;* (D3) hent tegn fra tastaturet
        move.l  #inline,d2      ;adresse på bufferen i D2
        move.l  dosbase,a6      ;DOS-basisadresse til A6
        move.l  conhandle,d1    ;vort vindue-handle
        jsr     Read(a6)        ;kald write-funktion
        rts                     ;færdig!

inline:         blk.b 80,0      ;buffer til tastaturindlæsning
```

Rutinen vender tilbage til hovedprogrammet, når der trykkes på return-tasten. Hvis der indlæses flere tegn end angivet i D3, bliver kun de første tegn overgivet til "inline". Når den kaldes en gang til, vender rutinen straks tilbage, og overgiver resten af de indlæste tegn.

Den form for indlæsning byder på megen komfort. Under indlæsningen kan arbejdes vilkårligt med backspace, da kun den linie, der til slut bliver tilbage, overgives til "inline". Desuden returneres antallet af anvendte tegn i D0.

Programmet afprøves på følgende måde:

Efter at have åbnet CON:-vinduet, skrives følgende linier i hovedprogrammet:

```
        move    #80,d3          ;tillad 80 tegn
        nsr     readchr         ;hent linie fra tastatur
        lea     inline,a0       ;adresse på linien i A0
        clr.b   0(a0,d0)        ;nulbyte i enden
        bsr     pmsg            ;udlæs linie igen

bp:
```

Herefter kommer den programdel, der atter lukker vinduet. Hvis man nu starter programmet med "g run" og angiver "bp" som breakpoint, så afbrydes programmet på dette sted. Således kan man betragte resultatet i ro og mag i vinduet. Derefter kan man fortsætte programmet med "j bp" og lade vinduet lukke, når programmet slutter med et RTS.

Efter programmets start og efter at vinduet er åbnet, fremkommer cursoren i vinduets øverste venstre hjørne. Nu indtastes en lille tekst og trykkes return. Man ser, at den tegnstreng, man lige har indtastet, udlæses omgående.

Udlæsningen blev iværksat med den i forrige kapitel præsenterede "pmsg"-rutine. Den kræver et nul sidst i den tekst, der skal udlæses. Nullet bliver sat på den indlæste tekst, ved at adressen på indlæsningsbufferen først bliver lagt ind i A0, og byte'n derefter slettes med CLR.B, der ligger på adressen A0+D0. Da antallet af tegn, der skal indlæses, ligger i D0, er byten altså den første ubenyttede byte.

Idet vi endnu befinder os i SEKA, når programmet afbrydes, kan man ved hjælp af "q inline" se på, hvad "readchr"-rutinen har returneret. Man genfinder de tegn, man har indtastet, plus et afsluttende $A. $A står for return-tasten og tælles med, så man efter indtastning af f.eks. "12" og return, får 3 som antal i D0.

Proceduren bør nu gentages med et RAW:-vindue. Derfor ændres vinduesdefinitionen CON: til RAW:, og programmet assembleres igen. Man bemærker forskellen. Der returneres nemlig straks efter det første tryk på en tast, og overgives derfor altid et 1 til D0.

Fordelen ved indlæsningen er, at cursor- og funktionstasterne bliver synlig. Man kan bearbejde disse specialtaster med en egnet rutine via gentagen indlæsning af et tegn med "readchr".

*Endnu en metode til tastaturindlæsning er aflæsningen af en enkelt tast. Det er f.eks. påkrævet, hvis et program beder om et tryk på "J"-tasten for "ja" ved et kontrolspørgsmål, når der skal udføres en vigtig funktion. Det kan også ske via en normal indlæsning, men er ofte bedre på denne måde.*

DOS-biblioteket indeholder en funktion, der venter et forudbestemt tidsrum på bekræftelsen med en tast, og returnerer så enten et nul (FALSE), hvis der ikke blev trykket på tasten, eller værdien -1 ($FFFFFFFF = TRUE), hvis der blev trykket. Hvilken tast, der blev trykket på, må bagefter vurderes. Funktionen, der har navnet WaitForChar, er kun praktisk, når der f.eks. under udlæsningen af en længere tekst, skal stoppes efter en side, og afvente et tasttryk, før næste side vises.

Funktionen kræver kun to parametre:

**I D1** Handle-nummeret på vinduet, resp. den fil, hvorfra tegnet skal læses. Der kan også ventes på et tegn fra et interface.

**I D2** overgives den tid i mikrosekunder, som skal ventes på et tasttryk, resp. modtagelse af et tegn.

For at vente et sekund på indlæsningen af en tast i vinduet, skal derfor skrives følgende rutine:

```
WaitForChar=-30-174

    ...
scankey:                       ;*vent på tasttryk
    move.l  conhandle,d1       ;i vinduet
    move.l  #1000000,d2        ;ventetid 1 sekund
    move.l  dosbase,a6         ;DOS-basisadresse
    jsr     waitforch(a6)      ;vent...
    tst.l   d0                 ;test resultat
    rts
```

Med TST-ordren i slutningen af programmet kan man, efter at have kaldt rutinen direkte udnytte resultatet, af den med en BEQ- eller en BNE-ordre: BEQ forgrener sig, hvis der ikke blev trykket på en tast, BNE derimod ikke.

## 6.4.3. Printer-tilgang

Efter at vi har diskuteret ind- og udlæsning via konsollen, vil vi udlæse data fra computeren. Det første apparatur, der egner sig til det, er printeren.

Udlæsninger på printer er meget nemme at foretage. Man skal blot åbne en kanal, som det allerede er kendt fra vinduerne. Kun skal der ikke angives CON: eller RAW: som navn, men PRT:.

Man åbner kanalen med de samme linier som før vinduet, men overgiver her en pointer på kanalnavnet "PRT:" i D1 og modus "ny" (1006) i D2 til "do_ open"-rutinen. Det derfra returnerede handle-nummer, lagrer man et eller andet sted, f.eks. med label'en "prthandle".

Man kan anvende de samme udlæsningsrutiner, som for vinduerne, til at udlæse tegn eller tekster på printeren. Man skal blot skrive "prthandle" i stedet for "conhandle" i linien "move.l conhandle,d1".

Det er endnu bedre at stryge denne linie. Så kan man anvende de samme rutiner, blot må værdien for vinduesudlæsning fra "conhandle" og for printerudlæsning fra "prthandle", overgives til rutinen i D1. På den måde opnår man en fleksibel udlæsningsrutine, der betjener såvel vindue som printer på samme måde. Selvfølgelig kan en indlæsning fra printeren ikke fungere, da printeren ikke kan sende, men kun modtage og printe data.

## 6.4.4. Seriel ind-/udlæsning

Det serielle interface er ligeså nemt at betjene som printeren, når man angiver SER: som filnavn. Den kanal, der åbnes dermed, tillader nu normale ind-

udlæsninger, der foretages med DOS-funktionerne READ og WRITE. Parametrene til interfacet som "handshake" eller overførelseshastighed kan indstilles med "preferences"-programmet.

## 6.4.5. Sprogudlæsning

Som noget særligt har Amigaen indbygget en sprogsynthesizer. Det er desværre ikke så let at programmere den, som de hidtil omtalte ind-/udlæsninger. Det er en såkaldt "device" med navnet "narrator.device".

Den kræver nogle programmeringstrin, til at installere og aktivere sig. Dertil hører åbningen af device'n, start af ind-/udlæsning osv. Jeg vil ikke gå ind på alle disse forhold i enkeltheder, men præsentere læseren for en rutine, der både oversætter en tekst fra den normale tekst og udlæser den.

Vi har brug for nogle initialiseringer. Hertil hører den almindelige definition af konstanter, hvortil kan tilføjes nogle ukendte. Her er de vigtigste, hvoriblandt der er forudset åbningen af et vindue.

```
;***** Narrator-grundfunktioner 3/87 S.D. *****

OpenLib    = -408
closelib   = -414
ExecBase   =  4

Open       = -30      ;åben fil
Close      = -36      ;luk fil
mode_old   =  1005    ;modus gammel
OpenDevice = -444     ;åben device
CloseDev   = -450     ;luk device

SendIo     = -462     ;start I/O
AbortIO    = -480     ;afbryd I/O

Translate  = -30      ;oversæt tekst
```

Herefter følger selve initialiseringen:

```
init:                         ;initialiser og åben system

;* Åbning af DOS-bibliotek *
```

```
        move.l  execbase,a6         ;pointer på EXEC-bibliotek
        lea     dosname,a1          ;pointer på DOS-navn
        moveq   #0,d0               ;version:ligegyldig
        jsr     openlib(a6)         ;åben DOS-library
        move.l  d0,dosbase          ;gem handle
        beq     error               ;i givet fald fejl

;* Åben oversætter-bibliotek (translator-library)

        lea     transname,a1        ;pointer på translator-navn
        clr.l   d0
        jsr     openlib(a6)         ;åben translator
        move.l  d0,tranbase         ;gem handle
        beq     error               ;i givet fald fejl

;* Sæt I/O-område for narrator *

        lea     talkio,a1           ;pointer på I/O-område i A1
        move.l  #nwrrep,14(a1)      ;indfør portadresse
        move.l  #amaps,48+8(a1)     ;pointer på audiomasker
        move    #4,48+12(a1)        ;antal masker
        move.l  #512,36(a1)         ;længde af udlæsningsområdet
        move    #3,28(a1)           ;kommando:skriv
        move.l  #outtext,40(a1)     ;adresse på udlæsningsområdet

;* Åben narrator-device *

        clr.l   d0                  ;nummer 0
        clr.l   d1                  ;flag: ingen
        lea     nardevice,a0        ;pointer på device-navn
        jsr     opendevice(a6)      ;åben narrator-device
        tst.l   d0                  ;fejl ?
        bne     error               ;ja!

;* Åben vindue *

        move.l  #consolname,d1      ;konsol-definition
        move.l  #mode_old,d2        ;modus: gammel
        move.l  dosbase,a6          ;DOS-basisadresse
        jsr     Open(a6)            ;åben vindue
        tst.l   d0                  ;fejl ?
        beq     error               ;ja !
        move.l  d0,conhandle        ;ellers gem handle
```

Efter vi har foretaget disse initialiseringer, kan vi lade den forberedte tekst fremsige. For også at kunne se, hvad det er, Amigaen fortæller os, kan vi samtidig lade teksten udlæse i vinduet med "pline"-rutinen:

```
        move.l  #intext,d2          ;tekst der skal siges
        bsr     pline               ;også udlæse i vindue

sayit:                              ;lade tekst sige

;* Oversætte tekst til lydskrift *

        lea     intext,a0           ;tekstens adresse
        move.l  #outtext-intext,d0  ;tekstens længde
        lea     outtext,a1          ;udlæsningsområdets adresse
        move.l  #512,d1             ;udlæsningsområdets længde
        move.l  tranbase,a6         ;translator-basisadresse
        jsr     Translate(a6)       ;oversæt tekst

;* Sprogudlæsning *

        lea     talkio,a1           ;I/O-strukturens adresse
        move.l  #512,36(a1)         ;udlæsningsområdets længde
        move.l  execbase,a6         ;EXEC-basisadresse
        jsr     SendIO(a6)          ;start I/O resp. sprogudlæsning
```

Idet vi stopper ind-/udlæsningen ved programmets afslutning, skal tilføjes noget, der "holder fast" i programmet. Hertil egner "readchr"-funktionen sig, som vi jo allerede har programmeret:

```
        bsr     readchr             ;vent på tastaturindlæsning
```

Computeren venter, indtil vi trykker på return-tasten. Vi kan altså i ro og mag høre på, hvad Amigaen har at sige. Trykkes så på return, bliver alt igen koblet fra, resp. lukket:

```
quit:
        move.l  execbase,a6         ;EXEC-basisadresse
        lea     talkio,a1           ;pointer på I/O-område
        jsr     abortio(a6)         ;afbryd I/O

        move.l  conhandle,d1
        move.l  dosbase,a6
        jsr     close(a6)           ;luk vindue
```

```
        move.l  dosbase,d1
        move.l  execbase,a6
        jsr     closelib(a6)        ;luk DOS-bibliotek

        lea     talkio,a1
        jsr     closedev(a6)        ;luk narrator-device

        move.l  tranbase,a1
        jsr     closelib(a6)        ;luk translator-bib.

        rts                         ;slut på programmet
```

Her følger de nødvendige data, til ovenstående program:

```
text:       dc.b 'Dette er en prøve-tekst !',10,13,10,13,0,0
dosname:    dc.b 'dos.library',0,0
transname:  dc.b "translator.library",0
consolname: dc.b 'RAW:0/100/640/100/** Test-vindue',0
nardevice:  dc.b 'narrator-device',0
even
dosbase:    dc.l 0
tranbase:   dc.l 0
amaps:      dc.b 3,5,10,12
even
conhandle:  dc.l 0
talkio:     blk.l 20,0
nwrrep:     blk.l 8,0
intext:     dc.b 'hello, i am the amiga talking to you',0
even
outtext:    blk.l 512,0
```

Det er lidt omstændeligt, men belønnes med de muligheder, der åbnes. Der findes nemlig en mængde varianter, som fåes ved at ændre nogle parametre. Parametrene er de indførelser i I/O-området, som ligger fra label'en "talkio". Området er opbygget således:

| Offset | Længde | Betydning |
|---|---|---|
| ** Port-data ** | | |
| 0 | L | pointer på næste blok |
| 4 | L | pointer på sidste blok |
| 8 | B | I/O-type |
| 9 | B | prioritet |

| Offset | Længde | Betydning |
|---|---|---|
| 10 | L | pointer på I/O-navn |
| 14 | L | pointer på port |
| 18 | W | længde |
| ** I/O-data ** | | |
| 20 | L | pointer på device |
| 24 | L | pointer på device-unit |
| 28 | W | kommandoord |
| 30 | B | I/O-flag |
| 31 | B | I/O-status |
| 32 | L | I/O-pointer |
| 36 | L | I/O-længde |
| 40 | L | pointer på data |
| 44 | L | I/O-offset |
| ** Egentlige narrator-data ** | | |
| 48 | W | talehastighed |
| 50 | W | stemmehøjde |
| 52 | W | sprogmodus |
| 54 | W | køn (mandlig/kvindelig stemme) |
| 56 | L | pointer på audio-masker |
| 60 | W | antal masker |
| 62 | W | lydstyrke |
| 64 | W | afspilningshastighed |
| 66 | B | flag til mundgrafik-frembringelse (0=slået fra) |
| 67 | B | aktuelle maske (intern anvendelse) |
| 68 | B | anvendte kanaler (intern anvendelse) |

*Det kan ikke anbefales at eksperimentere med dataene i de første to blokke, da det kan ende med et crash. Derimod er de sidste indførelser i denne struktur mere interessante, fordi vi kan opnå interessante effekter med dem.*

En oversigt over de parametre, hvormed vi kan variere sprogudlæsningen. Den værdi, der er angivet i parenteser, er standard-værdien, der indstilles ved åbningen af narrator.device.

**Talehastighed (150)**
Hermed kan man indstille den hastighed, der tales med. Stemmelejet forbliver stort set uforandret.

**Stemmehøjde (110)**
Man kan indskrive en værdi mellem 65 og 320 og således ændre stemmelejet (fra Fedtmule til Mickey Mouse).

**Sprogmodus (0)**
Nullet bevirker en halvejs naturlig taleform. Et ettal får Amigaen til at tale monotont, som en robot.

**Køn (0)**
Med et nul indstilles et mandligt stemmeleje, og med et ettal et kvindeligt (mere eller mindre...).

**Lydstyrke (64)**
Lydstyrken kan reguleres mellem 0 og 64. Standardlydstyrken er således den maksimale.

**Afspilningshastighed (22200)**
Når man sænker denne værdi, sænkes stemmen ligeledes. Ved stærke ændringer fremkommer nogle ret ejendommelige stemmer.

Man kan eksperimentere, indtil man har fundet den mest interessante stemme. Men hvis man lader en dansk tekst fremsige, vil man konstatere, at Amigaen har en hæslig udtale. Derfor må man, for at kunne lade en dansk tekst fremsige, ændre den til "amerikansk skrivemåde". For eksempel:

For at høre "dette er en dansk tekst", må man skrive "deddeh ehr en dansk tekst".

Det ser underligt ud, men lyder forståeligt. Prøv selv!

## 6.5. Disketteoperationer

Det vigtigste periferiudstyr, til en computer som Amigaen, er diskettestationen. På den gemmes alle de data, der ikke må gå tabt, når der slukkes for computeren. Hvordan man kan programmere denne lagring og naturligvis også genindlæsningen af dataene, skal vi se nærmere på i dette kapitel.

Vi ser først på de simple disketteoperationer, der tjener til administration af filer. For at kunne tilgå en fil, må vi først åbne den. Operationen kender vi en del til, fordi det er den samme OPEN-funktion fra DOS-biblioteket, som vi har brugt flere gange i det foregående kapitel. I de følgende eksempler går jeg ud fra, at DOS-biblioteket er åbnet.

## 6.5.1. Åbning af filer

Vedrørende OPEN-funktionen har vi fastslået, at der kræves en parameter for modus. Denne modus har en særlig betydning. Åbnes filen for at kunne læse fra den, er modus tilsyneladende allerede til stede. Modus skal derfor angives med "gammel" (1005).

Vil man oprette en fil, må man først åbne den. Da den i så fald slet ikke eksisterer endnu, overgives modus "ny" (1006). Hvis en fil åbnes til skrivning med denne modus, selv om den eksisterer under samme navn, så slettes den gamle fil med det pågældende navn, og derefter oprettes en ny. For at undgå tab af data, bør man derfor altid undersøge, om filen eksisterer, og i givet fald udlæse en fejlmelding.

Vi laver en underrutine, der åbner en fil. Vi går ud fra, at filnavnet står fra label'en "filename", og er afsluttet af en nulbyte. Så overgives modus i register D2 til underrutinen.

Rutinen afleverer det modtagne filehandle-nummer i "filehd" og vender tilbage til det hovedprogram, der foretog kaldet. Da operationen med handle er underprogrammets sidste, kan status vurderes lige derefter: hvis operationen er korrekt udført, og filen er åbnet, er handle-nummeret ulig nul. I modsat tilfælde er det nul, hvad der giver mulighed for, efter "bsr openfile", at forgrene direkte med "beq error" til en rutine, der eventuelt udlæser en fejlmelding.

Her er underprogammet til åbning af en fil, og samtidig et til at lukke en fil:

```
Open      = -30
Close     = -36
Mode_old  = 1005
Mode_new  = 1006

    ...
openfile:                         ;* åben fil, modus til D0
    move.l  dosbase,a6            ;DOS-basisadresse in A6
    move.l  filename,d1           ;pointer på filnavn
    jsr     Open(a6)              ;åben fil
    move.l  d0,filehd             ;gem handle
    rts

closefile:                        ;* luk fil
    move.l  dosbase,a6            ;DOS-basisadresse til A6
    move.l  filehd,d1             ;filehandle til D1
    jsr     Close(a6)             ;luk fil
    rts
```

```
filehd:     dc.l 0            ;plads til filehandle
filename:   dc.b "filnavn",0  ;fil der skal åbnes
 even
```

For at kunne anvende rutinerne, må vi se på, hvordan man kan load'e og save filer.

## 6.5.2. Læsning og skrivning af filer

Lad os begynde med at anlægge, og beskrive en ny fil. For at begynde, skriver vi først linierne:

```
move.l  #mode_new,d2  ;lagre nye fil
bsr     openfile      ;åben ny fil
beq     error         ;gik ikke!
```

Som filnavn skrives i linien med "filename" et navn, f.eks. "Testfil", der afsluttes med nulbyte. Efter at "openfile"-rutinen er kaldt, anlægges en fil med navnet på disketten; hvis der allerede eksisterer en sådan, slettes denne først.

Lad os antage, at vi har en tekst, som vi vil skrive i filen. Vi kan foregive en tekst ved at skrive:

```
text:       dc.b "Dette er en testtekst til tekstfilen",0
textende:
```

Label'en "textende" tjener til at fastsætte antallet af databytes ved subtraktion fra "text".

Teksten skrives i filen. Hertil anvender vi funktionen WRITE, der kræver tre parametre:

**I D1** Filehandle't, som vi har fået fra OPEN-funktionen,

**I D2** En pointer på de data, der skal skrives og

**I D3** Antallet af data, der skal skrives, i bytes.

Det betyder, at vi kan skrive endnu et underprogram, som skal overgives pointeren på dataene i D2 og antallet af data i D3:

```
Write       = -48

    ...
writedata:                       ;* skriv data i filen
    move.l  dosbase,a6           ;DOS-basisadresse til A6
    move.l  filehd,d1            ;filehandle i D1
    jsr     Write(a6)            ;skriv data
    rts
```

Underrutine knaldes, idet vi efter åbningen af filen skriver følgende linier i hovedprogrammet:

```
    move.l  #text,d2              ;pointer på dataene
    move.l  #textende-text,d3     ;antal bytes
    bsr     writedata             ;skriv data i filen
```

Derefter lukker vi filen med

```
    bsr     closefile            ;luk fil
    bra     ende                 ;afslut programmet
```

Hvis man nu assembler programmet med SEKA, starter det, og bagefter udlæser diretory'et med "v", så finder man den nye fil "Testfil", som har vor teksts længde. Filen indlæses på ny, for at fastslå om den indeholder de rigtige data.

Hertil bruger vi DOS'ets READ-funktion, der kræver de samme parametre som WRITE. Angivelsen af det antal bytes, der skal læses, kan anvendes til, kun at indlæse en del af filen. Hvis man angiver mere, end der er i filen, så load'es hele filen. Antallet af læste bytes finder man i register D0.

Vi anlægger nu et felt, hvor der er plads nok til de data, der skal læses. Det gøres på følgende måde:

```
felt:       blk.b                ;reserver 100 bytes
```

Til vort fileksempel er det tilstrækkeligt. Hvis man vil load'e en anden fil, skal man reservere mere plads.

Vi skriver endnu en underrutine til læsning af filen. Herunder går vi ud fra, at hele filen altid skal load'es. Vi behøver kun at forsyne underrutinen med adressen på den buffer, hvori dataene skal indlæses, i vort eksempel adressen "felt".

Her er underrutinen, der load'er hele den åbnede fil i det lagerområde, der blev angivet i D2:

```
READ      = -42

    ...
readdata:                         ;*udlæs fil
    move.l   dosbase,a6           ;DOS-basisadresse til A6
    move.l   filehd,d1            ;filehandle til D1
    move.l   #$ffffff,d3          ;læs et vilkårligt antal bytes
    jsr      Read(a6)             ;læs data
    rts
```

For at load'e den netop anlagte fil i den buffer, der er anlagt under "felt", ved hjælp af rutinerne, anvender vi følgende hovedprogram:

```
    move.l   #mode_old,d2 ;gamle fil
    bsr      openfile     ;åben fil
    beq      error        ;gik ikke!

    move.l   #felt,d2     ;pointer på databuffer
    bsr      readdata     ;indlæs fil
    move.l   d0,d6        ;gem byteantal i D6

    bsr      closefile    ;luk fil
    bra      ende         ;programmets ende
```

Efter assemblering og start af programmet, kan man med "q felt" se i databufferen, hvad man har indlæst fra filen. I D6 finder man desuden antallet af bytes, der blev læst fra filen.

## 6.5.3. Sletning af filer

Har man eksperimenteret nok med de ovenstående programmer, vil man gerne atter slette filen "Testfile" fra disketten. Dertil findes funktionen DELETEFILE i DOS-biblioteket, som har offset -42 og kun kræver n parameter. Parameteren overgives til D1 og udgør en pointer på filnavnet. Filnavnet skal igen afsluttes med en nulbyte.

For at slette filen med navnet "Testfile" kan man altså skrive følgende linier:

```
DeleteFile  = -42

    ...
    move.l   dosbase,a6       ;DOS-basisregister til D6
    move.l   #filename,d1     ;pointer på filnavn i D1
    jsr      DeleteFile(a6)   ;slet fil
```

Så er den fil væk. Filen kan ikke længere reddes med normale midler, hvis den skulle være blevet slettet ved en fejltagelse. Derfor bør man anvende et trick ved vigtige filer, for at gemme den gamle. Tricket anvendes også i mange programmer.

## 6.5.4. Ombenævnelse af filer

Når f.eks. et tekstbehandlingsprogram lægger en tekst, der er blevet ændret, tilbage på disketten, så bliver den gamle fil i reglen ikke slettet. Ofte bliver den gamle fil ombenævnt. F.eks. får den navnet "Backup", og den nye tekst bliver skrevet i den nye fil, med det gamle navn.

Den funktion i DOS-biblioteket, der varetager funktionen, hedder RENAME og har offset'et -48. Der overgives to pointere til den som parametre, nemlig i D1 pointeren på det gamle navn og i D2 pointeren på det nye.

For at omdøbe filen "Testfile" til "Backup" (før sletningen), anvendes følgende linier:

```
Rename      = -48

         ...
         move.l  dosbase,a6     ;DOS-basisadresse til A6
         move.l  #oldname,d1    ;pointer på det gamle navn i D1
         move.l  #newname,d2    ;pointer på det nye navn i D2
         jsr     Rename(a6)     ;ombenævn fil

         ...
oldname:  dc.b  "Testfile",0
newname:  dc.b  "Backup",0
```

## 6.5.5. Udlæsning af indholdsfortegnelse.

Man bedes forestille sig, at have programmeret en Text-Editor og startet dette program. Når man vil load'e en tekst fra disketten og bearbejde den, melder tvivlen sig, - hvad var det tekstfilen hed?

Der er brug for en funktion til at udlæse og vise en diskettes indholdsfortegnelse. Der er flere måder at gøre det på. Først den letteste måde, der ikke kræver megen programmering, men to diskettestationer eller disketteskift.

Hele tricket består i, selv at kalde det DIR- eller List-program, som indeholdes i C-directory'et, eller den tilsvarende kommando i CLI. DOS-biblioteket

indeholder en kommando hertil, ved navn "Execute", med offset'et -222, der gør der muligt at lade udføre en CLI-kommando.

Funktionen får tre parametre med på vejen:

**I D1** en pointer på den med nul afsluttede tekst, som angiver den kommando, der skal udføres. Teksten svarer nøje til den kommando, man ville indtaste i CLI'en. Også en nul-pointer er mulig.

**I D2** bestemmes indlæsningsfilen. Normalt indføres et nul. Men hvis man angiver file-handle't for en åbnet tekstfil, så bliver denne fil udlæst efter udførelsen af den forud givne kommando, og fortolket som kommandosekvens. Et vindue kan også defineres som indlæsningsmedium, hvormed man får et nyt CLI-vindue!

**I D3** står angivelsen af udlæsningsfil. Hvis der står nul, så fremkommer udlæsningerne af den kaldte kommando i standard-vinduet (CLI-vinduet).

For at prøve det igennem, skal man indtaste følgende underprogram i hovedprogrammet, hvori DOS-biblioteket og et vindue allerede er åbnet:

```
Execute  = -222

         ...
dir:
         move.l  dosbase,a6        ;DOS-basisadresse i A6
         move.l  #command,d1       ;pointer på kommandolinie
         clr.l   d2                ;ingen indtastning (CLI-vindue)
         move.l  windowhd,d3       ;udlæsning i vort vindue
         jsr     Execute(a6)       ;udfør kommando
         rts

command:
         dc.b  "dir",0
```

Programmet fungerer også med List-kommandoen. Manglen ved den metode vist, at enten skal den diskette, som Workbench blev load'et med, være i stationen, eller også skal den lægges i på opfording fra systemet. DIR-kommandoen er bare et program, som Amigaen skal load'e, før det kan udføres.

Man skal ikke se bort fra fordelene ved ovenstående metode, programmet er dejligt kort, og giver desuden mulighed for at indbygge en vilkårlig kommando fra CLI'en i et program.

Vi vil se bort fra fordelene og finde på en metode, der ikke kræver CLI'ens medvirken. Så kan vi også udlæse indholdsfortegnelsen af alle mulige disketter uden at skulle jonglere rundt med disketterne.

Hvad vi skal programmere, er det, som CLI's DIR-program også kan gøre. Det er flere trin:

Vi må give systemet en nøgle til det ønskede directory. Det betyder, at vi skal kalde DOS'ets Lock-funktion, som vi overgiver to parametre:

**I D1** overgiver vi en pointer på en tekst, der bestemmer, hvilken indholdsfortegnelse, der skal indlæses. Teksten svarer til DIR-funktionen; når vi f.eks. vil have indholdet af RAM-disken, lyder den "RAM:"0.

**I D2** står den modus, der bestemmer, om der skal læses eller skrives. Vi vælger her modus "læse" (-2).

Dermed kalder vi Lock-funktionen med offset'et -84 og får enten returneret en pointer på nøglen, eller et nul i D0. Et nul angiver, at kaldet er mislykkedes, resp. at filen ikke blev fundet. Vi kan altså fastslå ved hjælp af funktionen, om en fil er på disketten, idet vi blot overgiver filens navn, og tester om der returneres et nul i D0. Hvis det ikke er tilfældet, så eksisterer filen.

Lad os gå ud fra, at filen, resp. vejen eksisterer. Så skal den returnerede værdi i D0 lagres. Den behøves af de to funktioner, som vi mangler at kalde.

Den funktion, der kommer herefter, har navnet Examine. Med den gennemsøges disketten efter den første passende indførelse, og de parametre, der hører til indførelsen såsom navn, længde eller dato returneres. Vi skal reservere et passende lagerområde til funktionerne og overgive det begyndelse til Examine-funktionen i D2. Til D1 overgives nøglen, som Lock-funktionen har givet.

*Lagerområdet, fyldes med information, kaldet FileInfoBlock. Det er på 260 bytes og indeholder alle informationer om filen. Navnet ligger fra den 9. byte og ender med en nul-byte, så vi nemt kan udlæse den med "pmsg"-rutinen. De informationer, som Examine udlæser, gælder ikke for en fil, men for disketten. Navnet i FileInfoBlock'en vil således være diskettenavnet.*

Examine-funktionen returnerer i D0 den status, som angiver, om filen er fundet. Da den forudgående Lock-funktion har testet det, er brugen af denne status egentlig unødvendig.

Nu til den funktion, hvormed vi kan udlæse de enkelte filer fra indholdsfortegnelsen. Funktionen bærer det betegnende navn ExNext, hvad der betyder Examine Next. Med funktionen søges, ved hvert kald, efter den næste indførelse, som nøglen passer på. Som parametre får ExNext overgivet de samme data som Examine. Returparameteren i D0 spiller dog en langt større rolle end før.

ExNext-funktionen kaldes altid på samme måde, og opgiver altid det næste felt i directory'et; men findes der ikke nogen passende felt, så giver ExNext et nul i D0.

Så skulle vi være færdige med indholdsfortegnelsen. Men for at afslutte på en pæn måde, bør vi se efter, om den sidste operations status virkelig lyder "ikke flere felter". Det undersøges med DOS's IoErr-funktion.

Funktionen kræver ingen parametre. Den returnerer status af sidst udførte I/O-operation i D0. Efter den sidste ExNext skal den være 232, hvad der betyder no_more_entries.

Nu en fuldstændig rutine, der udlæser indholdsfortegnelsen for disketten i station DF0: i det vindue, hvis handle står i "windowhd", og bagefter returnerer status'en (232=$E8) i D6:

```
Lock     = -30-54
Examine  = -30-72
ExNext   = -30-78
IoErr    = -30-102

    ...
directory:                          ;* indholdsfortegnelse for DF0:
    move.l  #titel,d0
    bsr     pmsg                    ;udlæs overskrift

    move.l  dosbase,a6              ;DOS-basisadresse til A6
    move.l  #name,d1                ;pointer på vej-/filnavn
    move.l  #-2,d2                  ;'læs' modus
    jsr     Lock(a6)                ;søg fil
    tst.l   d0                      ;fundet?
    beq     error                   ;nej!
    move.l  d0,locksav              ;ellers gem nøgle

    move.l  dosbase,a6              ;DOS-basisadresse
    move.l  locksav,d1              ;nøgle til D1
    move.l  #fileinfo,d2            ;pointer på FileInfoBlock
    jsr     examine(a6)             ;hent disk-navn
    tst.l   d0                      ;OK?
    beq     error                   ;nej (forekommer næppe)
    bra     udlæs                   ;ellers udlæs navn

loop:                               ;* udlæs filnavn
    move.l  dosbase,a6              ;DOS-basisadresse
    move.l  locksav,d1              ;nøgle til D1
```

```
        move.l  #fileinfo,d2        ;pointer på FileInfoBlock
        jsr     ExNext(a6)          ;søg næste fil
        tst.l   d0                  ;fundet
        beq     error               ;nej: slut

udlæs:                              ;* udlæs navn
        move.l  #fileinfo+8,d0      ;pointer på navn
        bsr     pline               ;udlæs navn
        bra     loop                ;og gå videre...

error:                              ;* find ud af I/O-status
        move.l  dosbase,a6          ;DOS-basisadresse til A6
        jsr     IoErr(a6)           ;hent status
        move.l  d0,d6               ;og gem i D6
        rts                         ;slut...

navn:   dc.b 'DF0:',0
even
locksav: blk.l  0
fileinfo: blk.l 260
```

De felter, der skal indføres i FileInfo blokken er følgende:

| Offset | Navn | Betydning |
|---|---|---|
| 0 | DiskKey.L | Diskettenummer |
| 4 | DirEntryType.L | Optegnelsestype (+ = directory, - = fil) |
| 8 | Filnavn | 108 bytes med filnavn |
| 116 | Protection.L | Fil beskyttet? |
| 120 | EntryType.L | Optegnelsestype |
| 124 | Size.L | Fillængde i bytes |
| 128 | NumBlocks.L | Antal blokke, der optages |
| 132 | Days.L | Oprettelsesdato |
| 136 | Minute.L | Oprettelsestidspunkt |
| 140 | Tick.L | Oprettelsestidspunkt |
| 144 | Comment | 116 bytes med kommentar |

Når man foruden navnet vil udlæse fillængden i ovenstående program, kan man igangsætte det, ved at udlæse længden med "move.l fileinfo+124,d0" og et påfølgende kald af forvandlingsrutinen, der frembringer et decimaltal heraf. Resultatet kan udlæses sammen med navnet.

## 6.5.6. Direkte tilgang til disketten

Den direkte tilgang til enkelte sektorer på disketten, indeholdes ikke som en simpel funktion i bibliotekerne. Her må vi, ligesom ved sprogudlæsningen, arbejde med en device. Denne gang hedder device'n Track-disk.device.

Med devicen afprøves den direkte programmering af diskettestation(er). Når vi har fået opbygget et programskelet, kan vi afprøve de enkelte kommandoer til diskettetilgang.

*Bemærk, at en fejl herunder, kan modificere den indlagte diskette under kørslen, og gøre den uanvendelig. Som "forsøgskanin" bør man derfor altid anvende en diskette, der ikke indeholder nogen vigtige data.*

Programmeringen begynder med en lignende initialisering som ved sprogudlæsningen. Her er programmets initialiseringsrutiner

```
;** direkte diskettetilgang via Trackdisk.device **

OpenLib=-408
closelib=-414
ExecBase=4

Open=-30
Close=-36
opendevice =-444
CloseDev =-450
SendIo =-462
Read=-30-12
Write=-30-18
WaitForCh=-30-174
mode_old=1005

run:
    bsr     init            ;initialisering
    bra     test            ;system-test

init:                       ;system initialisering og åbning
    move.l  execbase,a6     ;pointer på EXEC-bibliotek
    lea     dosname,a1
    moveq   #0,d0
    jsr     openlib(a6)     ;åben DOS-library
    move.l  d0,dosbase
    beq     error
```

```
        lea     diskio,a1            ;pointer på disk-I/O-område
        move.l  #diskrep,14(a1)      ;pointer på port
        clr.l   d0                   ;station 0 (indbygget)
        clr.l   d1                   ;ingen flag
        lea     trddevice,a0         ;pointer på device-navn
        jsr     opendevice(a6)       ;åben trackdisk.device
        tst.l   d0                   ;fejl?
        bne     error                ;ja!

        move.l  #consolname(pc),d1 ;consol-definition
        move.l  #mode_old,d2         ;modus: gammel
        move.l  dosbase,a6           ;DOS-basisadresse
        jsr     open(a6)             ;åben vindue
        tst.l   d0                   ;fejl?
        beq     error                ;ja!
        move.l  d0,conhandle         ;ellers gem handle

        rts                          ;færdig

test:                                ;plads til test-rutiner
```

Og så også de nødvendige funktioner, der sørger for de forskellige afmeldinger ved programmets afslutning:

```
error:
        move.l  #-1,d7               ;flag til fejl (til SEKA)
qu:
        move.l  execbase,a6          ;EXEC-basisadresse
        lea     diskio,a1            ;pointer på disk-I/O
        move.l  32(a1),d7            ;IO ACTUAL til D7 (til testning)
        move    #9,28(a1)            ;kommando: motor tændt/slukket:
        move.l  #0,36(a1)            ;0=slukket, 1=tændt, sluk altså
        jsr     sendio(a6)           ;motor

        move.l  conhandle,d1         ;luk vindue
        move.l  dosbase,a6
        jsr     close(a6)
        move.l  dosbase,d1           ;luk DOS-bibliotek
        move.l  execbase,a6
        jsr     closelib(a6)

        lea     diskio,a1
        jsr     closedev(a6)         ;luk trackdisk.device

        rts
```

Man må ikke glemme den rutine, der venter på et tryk på return-tasten, for at vi kan afvente testfunktionens effekt i ro og mag:

```
getchr:                              ;* hent et tegn fra tastaturet
        move.l  #1,d3                ;1 tegn
        move.l  conhandle,d1         ;vindueshandle
        move.l  #inbuff,d2           ;buffer-adresse
        move.l  dosbase,a6           ;DOS-basisadresse
        jsr     read(a6)             ;indlæs tegn
        rts                          ;det var det
```

Til slut de datafelter og tekster, som programmet behøver:

```
dosname:    dc.b 'dos.library',0
even
consolname dc.b 'RAW:0/100/640/50/** vente-vindue',0
even
trddevice:  dc.b 'trackdisk.device',0
even
dosbase:    dc.l 0               ;DOS-basisadresse
conhandle:  dc.l 0               ;vindueshandle
inbuff:     blk.b 80,0           ;tastatur-buffer
diskio:     blk.l 20,0           ;I/O-struktur
diskrep:    blk.l 8,0            ;I/O-port
diskbuff:   blk.b 512*2,0        ;plads til 2 sektorer
```

Det var pligtarbejdet, så kommer vi til det mere fornøjelige og, hvordan man kan videregive kommandoer til diskettestationen. Den første kommando, der samtidig er den simpleste, er allerede dukket op: tænding og slukning af stationens motor. Nummeret på kommandoen er 9. Nummeret indføjes i I/O-strukturens kommandoord, der ligger i 28/29. byte af strukturen.

Der må overgives endnu en parameter for at bestemme, om motoren skal være tændt eller slukket. Informationen skal i I/O-længde-langordet fra den 36. byte, nemlig som 0 for slukning og 1 for tænding af motoren.

Hvilken motor, der skal tændes eller slukkes, angives når device'n åbnes. I D0 overgives den valgte stations nummer, her er det 0 for den indbyggede station DF0.

En oversigt over kommandoer, hvormed vi kan tilgå disketten:

| Nr. | Navn | Funktion |
|---|---|---|
| 2 | **READ** | **Læsning af en eller flere sektorer** |
| 3 | **WRITE** | **Skrivning af sektorer** |
| 4 | **UPDATE** | **Tilbageskrivning af track-bufferen** |
| 5 | **CLEAR** | **Sletning af track-bufferen** |
| 9 | **MOTOR** | **Tænd eller sluk for motor** |
| 10 | **SEEK** | **Søg track** |
| 11 | **FORMAT** | **Formatter tracks** |
| 12 | **REMOVE** | **Initialiser den rutine, der kaldes, når disketten fjernes** |
| 13 | **CHANGENUM** | **Fastslå antallet af disketteskift** |
| 14 | **CHANGESTATE** | **Test, om en diskette er indlagt** |
| 15 | **PROTSTATUS** | **Test, om disketten er skrivespærret** |

Kommando nr. 9 har vi lært. Lad os se på de tre kommandoer, man kan stille spørgsmål med. Det er de sidste 3 kommandoer, der lægger en returværdi i det lange ord fra den 32. byte i I/O-strukturen. Til testning bliver værdien i ovenstående program skrevet i register 7, hvis indhold man kan aflæse med SEKA efter programmets kørsel.

Her er rutinen, hvormed en af disse kommandoer udføres:

```
test:
        lea     diskio,a1       ;pointer på I/O-struktur
        move    #13,28(a1)      ;overgiv kommando (f.eks. 13)
        move.l  execbase,a6     ;EXEC-basisadresse til A6
        jsr     SendIO(a6)      ;kald funktion
```

Når man overgiver CHANGENUM (13) som kommando, får man i D7 et tal tilbage, der angiver, hvor ofte en diskette blev lagt i, og taget ud af en station. Når man kalder programmet, får man altså værdien retur. Hvis man tager disketten ud, og sætter den ind igen, vil tallet være 2 højere ved et fornyet kald.

Kommandoen CHANGESTATE (14) giver information om, hvorvidt der er indlagt en diskette. Hvis ja, så returneres et 0, i modsat fald værdien $FF.

Den samme værdi får man også retur fra PROTSTATUS-funktionen (a5). Her betyder et nul, at disketten ikke er skrivebeskyttet, $FF at den er.

Så til skrive-/læsefunktionerne READ og WRITE. Operationerne kræver naturligvis nogle parametre mere end statusfunktionerne. Man må angive:

Adressen på ind/udlæsningsbufferen til datapointeren, antallet af databytes, der skal overføres, til I/O-længde og dataadressen på disketten til I/O-offset.

Antallet af databytes skal være et heltals-multiplum af 512, da hver sektor er 512 bytes lang, og der kun kan læses hele sektorer.

Datadressen betyder nummeret på den første byte på disketten, hvorfra dataene skal overføres. Hvis den første sektor skal anvendes, er offset'et altså 0. For den anden sektor er det 512 osv., dvs.:

**offset=(Sektornummer-1)*512**

Her er den rutine, hvormed de første to sektorer kan indlæses i bufferen:

```
test:
        lea     diskio,a1
        move    #2,28(a1)         ;command: READ
        move.l  #diskbuff,40(a1)  ;buffer
        move.l  #2*512,36(a1)     ;længde: 2 sektorer
        move.l  #0*512,44(a1)     ;offset: 0 sektorer
        move.l  execbase,a6       ;EXEC-basisadresse
        jsr     SendIO(a6)        ;start funktion
```

Hvis man starter dette program, kan man se indholdet af bufferen med "q puffer". Her kan man se, hvilket format disketterne formatteres med. Hvis man vil læse beskrevne sektorer, skal man ændre 0 i offset-definitionen til 700 og starte på ny. Her ligger med stor sandsynlighed nogle data.

Hvis man vil ændre de data, som man har hentet fra disketten, og skrive tilbage på disketten igen, skal man anvende WRITE-kommandoen (3). Parametrene forbliver de samme. *Men pas på: forsøg ikke at ændre dataene med SEKA's "m"-funktion og atter at assemble programmet med den ændrede disk-kommando. Så bliver bufferen nemlig fyldt med nuller, og alle data er væk!*

Har man alligevel givet WRITE-kommandoen, undres man måske over, at diskettestationen forbliver inaktiv. Det skyldes, at Amiga'en skriver et læst spor i en speciel buffer og skriver de nye data i den, når den får en WRITE-kommando. Først når der skal gås til et andet spor, skrives disse data igen på disketten.

Man kan også lade tilbageskrivningen af dataene udføre direkte. Hertil bruges kommandoen UPDATE (4).

Endnu en interessant kommando er FORMAT (11). Kommandoen kræver et datafelt, der er 11*512=5632 bytes langt, altså ligeså langt som de data, hele sporet kan optage. Desuden skal angivelsen af offset'et være et multiplum af dette tal, for at begynde ved begyndelsen af et spor.

Angivelsen af længden må sluttelig være et multiplum af 5632. Hvis flere spor skal formatteres, fyldes hvert spor med de samme data.

Med funktionen kan man nemt skrive et diskettekopieringsprogram, der delvis indlæser indholdet af kildedisketten i lageret med READ, og så, efter at måldisketten er lagt i, udfører en formatteringskommando for hvert spor, hvor adressen på de respektive data hver gang angives i lageret. Således virker DiskCopy-programmet, som pricipielt nyformatterer måldisketten.

Kommandoen SEEK (10) kræver blot angivelse af offsets. Så kører stationens skrive-/læsehoved hen til den opgivne position, men uden at udføre nogen tilgang til disketten, eller en test for korrekt position.

REMOVE-kommandoen anvendes til installering af en interrupt-rutine, der kaldes fra stationen, når disketten fjernes. Interrupt-strukturens adresse overgives til I/O-strukturens datapointer. Hvis der indføres et nul, så frakobles interrupt-rutinen.

# 7. Arbejdet med Intuition

Efter at have lært så meget om maskinsprog, vil vi se på nogle specielle forhold ved Amiga'en. Hermed menes operativsystemet "Intuition", der klarer styringen af vinduer, skærmbilledformater (screens), musen og meget mere. Før vi programmerer denne Amiga-lækkerbidsken, har jeg en god og en dårlig oplysning til læseren.

Først den gode: Intuition har så mange funktioner og muligheder, at man har en utrolig fleksibilitet i programmeringen af sine ideer. Men nu til manglen derved: for at opnå den store fleksibilitet, skal man forberede og overgive en mængde parametre, som kræver et ret stort arbejde.

*Men der er ingen grund til panik: når man har bygget det nødvendige "skelet", dvs. programmeret det, bliver programmeringen og afprøvningen mere og mere interessant. Man bør altid kopiere sit program, før man afprøver nye varianter, da Intuition ofte reagerer følsomt på falske parametre og giver sig til at meditere over, hvad man egentlig mener...*

Men til værket. For at arbejde med Intuition, behøver vi Intuitions-biblioteket, som load'es med EXEC-funktionen OpenLibrary på samme måde som f.eks. DOS-biblioteket. Her er den tilsvarende programdel, der næppe behøver særlig omtale.

```
OpenLib   = -408
ExecBase  =  4

run:
        bsr     openint         ;load Intuitionsbibliotek
        ...

openint:                        ;* initialiser og åben system
        move.l ExecBase,a6      ;EXEC-basisadresse
        lea     IntName,a1      ;navn på Intuition-bibliotek
        jsr     OpenLib(a6)     ;åben Intuition
        move.l d0,intbase       ;gem Intuitions-basisadresse
        rts

IntName:  dc.b "intuition.library",0
even
intbase:  dc.l 0                ;basisadresse for intuition
```

Når programmet er kørt, skal biblioteket lukkes efter arbejdets afslutning, og efter den eventuelle lukning af Screens og vinduer. Hertil bruger man Close-Libray-funktion i EXEC med offset -414. Her er den tilsvarende underrutine:

```
CloseLibray = -414

    ...
closeint:                              ;luk Intuition
        move.l  execbase,a6            ;EXEC-basisadresse til A6
        move.l  intbase,a1             ;Intuitions-basisadresse til A1
        jsr     CloseLibrary(a6)       ;luk Intuition
        rts                            ;færdig
```

Når alt er klappet, kan vi vove det første skridt med Intuition.

## 7.1. Åbning af skærmbillede

Intuition er et grafisk operativsystem. Det er beregnet på, at kunne operere med skærmbilledet. Interessant nok kan det arbejde med flere skærmbilleder på samme tid, selv om man kun kan tilslutte n monitor til Amiga'en.

Løsningen mysteriet er de såkaldte screens, hvad der på dansk betyder billedskærme. Man kan åbne vilkårligt mange (så mange, som lageret strækker til) af den slags screens, hvorpå der kan åbnes vinduer, repræsenteres menuer og foretages ind-/udlæsninger. De enkelte screens er uafhængige af hinanden, og lader sig alle bearbejde samtidig på samme monitor. Problemet er blot, at man kun kan se, hvad der ligger "forrest" på billedskærmen.

Man kan sætte de enkelte screens foran eller bagved de andre, resp. skubbe dem op og ned. Desuden kan man, ved at trykke på den venstre Amiga-tast og "m" få workbench'ens screen forrest, hvis man er løbet sur i stablen af screens.

Vi begynder med programmeringen af intutition, idet vi anlægger vor egen screen. Vi har indlæst Intuitions-biblioteket, og behøver nu blot at kalde funktionen OpenScreen

*Men et øjeblik! Hvordan skal den screen se ud? Hvor skal den hen, og hvilken opbygning skal den have? Vi må se på, hvilke muligheder vi har for at udforme screens.*

Anvisningerne om, hvilken screen der skal åbnes, gives i form af en tabel, der har 13 parametre. Lad os betragte hver af disse parametre, der bestemmer vor screen.

Vi begynder tabellen med label'en "screen_defs", der skal stå på en lige adresse:

```
even
screen_defs:                    ;* her begynder screen-tabellen
```

De første anvisninger, som vi kan give vor screen, er position og størrelse. Vi siger altså, at den skal begynde helt foroven til venstre og fylde hele billedskærmen. Det giver position x=0 og y=0, bredden 320 og højden 200. Disse angivelser skal forstås i pixels, så vor screen får den maksimale størrelse.

```
x_pos:    dc.w    0   ;x-position
y_pos:    dc.w    0   ;y-position
width:    dc.w  320   ;bredde
heigth:   dc.w  200   ;højde
```

Dernæst kommer overvejelsen af, hvor mange farver der skal kunne repræsenteres. Det bestemmes af antallet af "bitplanes", også kaldet dybde. Hvis vi vælger 2, så kan ialt repræsenteres 2↑2, altså 4 farver, med 1 kun 2 farver. Lad os til en start vælge 2, da fire farver almindeligvis er nok.

```
depth:      dc.w    2   ;antal bitplanes
```

Nu bestemmer vi farverne, som titellinie og funktionssymboler skal tegnes med, idet vi angiver nummeret på farveregisteret:

```
detail_pen: dc.b    0   ;tekstens farve osv.
```

Og nu tillige tekst-baggrundens farve

```
block_pen:  dc.b    1   ;baggrundsfarve
```

Læg mærke til, at begge anvisninger skal foreligge i bytestørrelse. Farverne er, hvis de ikke er ændret, normalt de følgende (bemærk, at antallet af farver afhænger af antallet af bitmaps!):

| **Pen** | **Farve** |
|---|---|
| 0 | baggrund (blå) |
| 1 | hvid |

**med 2 bitplanes:**

| | |
|---|---|
| 2 | sort |
| 3 | rød |

**med 3 bitplanes:**

| | |
|---|---|
| 4 | blå |
| 5 | violet |
| 6 | tyrkis |
| 7 | hvid |

**med 4 bitplanes**

| | |
|---|---|
| 8 | sort |
| 9 | rød |
| 0 | grøn |
| 11 | brun |
| 12 | blå |
| 13 | blå |
| 14 | grøn |
| 15 | grå |

Nu følger et ord, i hvilket de enkelte bits beskriver screen'ens udseende. Disse bits er:

| Bit | Værdi | Navn | Betydning |
|---|---|---|---|
| **1** | **2** | **GENLOCK VIDEO** | |
| **2** | **4** | **INTERLACE** | **Omstiller screen'en til interlace funktionsmåde. Opløsningen og dermed de maksimale værdier for screenstørrelse fordobles på denne måde.** |
| **6** | **$40** | **PFBA** | |
| **7** | **$80** | **EXTRA HALFBRITE** | |
| **8** | **$100** | **GENLOCK AUDIO** | |
| **10** | **$400** | **DBLPF** | **Bevirker opdeling af en screen i ramme- og tegnområder** |
| **11** | **$800** | **HOLDNMODIFY** | **Indkobler Hold-and-Modify modus** |
| **13** | **$2000** | **VP HIDE** | |
| **14** | **$4000** | **SPRITES** | **Muliggør anvendelse af sprites** |
| **15** | **$8000** | **MODE 640** | **Omstiller screen'en til højopløsningsgrafik (640x400)** |

Til vor eksempel-screen vælges værdien 2 (normal):

**view_modes: dc.w 2** ;repræsentations-modus

Det ord, der følger, er igen opbygget således, at mange bits har en speciel betydning. Med ordet vælges, hvilken type screen. Hvis vi skriver 15, så bliver vor screen en "customscreen", hvad der giver os alle muligheder.

**screen_type: dc.w 15** ;screen-type: customscreen

Nu følger en pointer på det tegnsæt, der skal anvendes til alle udlæsninger på denne screen. Vil man ikke installere noget specielt tegnsæt, er det tilstrækkeligt, hvis man sætter et nul, så anvendes standard-tegnsættet.

**font: dc.l 0** ;tegnsæt: standard

Så mangler kun en pointer, som peger på den tekst, der skal være overskrift på screen'en. Teksten skal afsluttes med et nul, som vi kender det fra vindues-overskrifter.

**title: dc.l titel** ;pointer på titel-tekst

Dernæst følger et langt ord, hvormed de såkaldte "gadgets" defineres. Gadgets er funktionssymboler, der f.eks. kan klikkes på, for at bringe screen'en forrest. Det langord, der skal indføres i denne tabel, udgør en pointer på en liste, hvori gadgets optælles. Her er system-gadgets ikke medanført. Vi vil kun benytte system-gadgets, så at vi skal skrive et nul.

**gadgets: dc.l 0** ;ingen gadgets

Til allersidst kommer endnu et langt ord, som vi kun har brug for, hvis vi vil anvende en særlig bitmap til vor screen. Da det ikke er tilfældet, skriver vi også et nul her.

**bitmap: dc.l 0** ;ingen bitmap

Det var de listeindførelser, som behøves til definition af vor screen. Hvad der mangler, er overskriften, som i vort eksempel kan programmeres således:

**titel: dc.b 'Vor screen',0** ;screen-overskrift

Her er hele listen på en gang:

```
even
screen_defs:                          ;* her begynder screen-tabellen
x_pos:          dc.w    0             ;x-position
y_pos:          dc.w    0             ;y-position
width:          dc.w    320           ;bredde
heigth:         dc.w    200           ;højde
depth:          dc.w    2             ;antal bitplanes
detail_pen:     dc.w    0             ;farve af tekst etc.
block_pen:      dc.b    1             ;baggrundsfarve
view-modes:     dc.w    2             ;afbildnings-modus
screen_type:    dc.w    15            ;screen-type: customscreen
font:           dc.l    0             ;tegnsæt:standard
title:          dc.l    titel         ;pointer på titel-tekst
gadgets:        dc.l    0             ;ingen gadgets
bitmap:         dc.l    0             ;ingen bitmap

titel:          dc.b    'Vor screen',0 ;screen-overskrift
```

Med de således forberedte parametre, er det nemt at åbne screen'en. Hertil behøver vi Intuitions OpenScreen-funktion, der har offset -198, og blot får en parameter, nemlig adressen på parametertabellen ovenfor. Den resulterende programdel lyder derfor:

```
Openscreen = -198

        bsr     openint               ;åben intuition
        bsr     scropen               ;åben screen
        ...

scropen:                              ;* åben screen
        move.l  intbase,a6            ;Intuitionsbasisadresse til a6
        lea     screen_defs,a0        ;pointer på tabel
        jsr     openscreen(a6)        ;og åben
        move.l  d0,screenhd           ;gem screen'ens handle
        rts                           ;returner til hovedprogram

        ...

screen_defs:                          ;tabel se ovenfor
```

Og straks er Amigaens billedskærm dækket af vor egen screen. Nu kan der foretages alle mulige ting med den, indtil programmet har afsluttet sit arbejde; men derefter må scren'en straks lukkes, så man igen kan se workbench-screen'en.

Til lukning af skærmen anvendes CloseScreen-funktionen med offset -66, som vi overgiver den pointer på screen-strukturen, vi modtog fra OpenScreen-funktionen:

```
CloseScreen = -66

...
scrclose:                          ;* luk screen
        move.l intbase,a6          ;Intuitions-basisadresse til a6
        move.l screenhd,a0         ;screen-handle til A0
        jsr    CloseScreen(a6)     ;luk screen
        rts                        ;færdig
```

Lad os nu betragte det lange ord, der har returneret Open Screen-funktionen. Det er en pointer, og peger på en screen-struktur, der indeholder alle nødvendige data om screen'en. Foruden de af os angivne data, er der pointere på skærmbilledområder for de enkelte bitplanes osv.

Opbygningen af strukturen er kompliceret og indeholder nogle data, man ikke kan bruge til noget. Nogle af indførelserne er ret interessante. Her er et udvalg af de anvendelige parametre.

| Nr. | Navn | Funktion |
|---|---|---|
| **0** | **(NextScreen.L)** | **Pointer på den næste screen** |
| **4** | **(FirstWindow)** | **Pointer på den første vindue-struktur** |
| **8** | **(LeftEdge.W)** | |
| **$A** | **(TopEdge.W)** | **Screen'ens position** |
| **$C** | **(Width.W)** | **Bredde** |
| **$E** | **(Heigth.W)** | **Højde** |
| **$10** | **(MouseY.W)** | |
| **$12** | **(MouseX.W)** | **Musens position på skærmen** |
| **$14** | **(Flags.W)** | **Screen-flags** |
| **$16** | **(Title.L)** | **Pointer på titeltekst** |
| **$1A** | **(DefaultTitle)** | **Pointer på normal-titel** |
| **$28** | **(Font.L)** | **Pointer på tegnsæt** |
| **$C0** | **(plane0.L)** | **Pointer på bit-plane 0** |
| **$C4** | **(Plane1.L)** | **Pointer på bit-plane 1** |
| **$C8** | **(Plane2.L)** | **Pointer på bit-plane 2** |
| **$CC** | **(Plane3.L)** | **Pointer på bit-plane 3** |

Et eksempel på anvendelse af plane-pointere er, at skrive nogle tegnrutiner og bruge dem. Efter at have åbnet screen'en skal man indlæse adressen på et plane i et adresseregister med følgende kommandoer:

```
        move.l  screenhd,a5     ;screen-pointer til A5
        move.l  $c0(a5),a5      ;bit-plane 0-pointer til A5
```

Når man vil afprøve det, kan man forsøge at skrive følgende foran programmets "loop"-label:

```
        move.l  screenhd,a5     ;screen-pointer til A5
        move.l  $c0(a5),a5      ;bit-plane 0-pointer til A5
        move    #$20,d0         ;tæller D0=$20
lopl:
        move    d0,(a5)         ;skriv tællerbits i billede
        add.l   #80,a5          ;adresse +80, næste linie
        dbra    d0,lopl         ;videre indtil D0<0
```

Programdelen tegner et lodret, hvidt mønster i det øverste venstre hjørne af screen'en, der svarer til bitmønstret i tallene $20 til 0. Det er, må jeg indrømme, ikke noget særligt smart program, men det anskueliggør, hvor let det er, at skrive direkte i billedet. Hvis man ændrer offset'et i den anden linie til $C4, udskrives mønstret i rødt.

Man kan forskyde hele screen'en med den kendte teknik, hvor man anbringer musens pointer på den øverste kant, og skubber op og ned med venstre musetast nedtrykket. Men hvad musen kan, kan vi også med et program: denne forskydning er også mulig fra programmet.

Se på forskydningen af skærmen, uden mus. Til demonstration tager vi joystick'et, som tilsluttes til port 2. Som det blev forklaret i kapitlet om hardware-registre, kan stillingen af joysticket udlæses fra lagercelle $DFF00C. Vi kan i programeksempelet se, i hvilken retning screen"en skal bevæges.

Forskydningen iværksættes gennem endnu en Intuitions-funktion. Det er om funktionen MoveScreen med offset'et -162, som kræver tre parametre:

**I A0** pointeren på screen-strukturen, som vi får i D0 efter åbning af screen'en, og har gemt i "screenhd".

**I D1** den ønskede forskydning i y-retningen, altså vertikalt.

**I D0** den horisontale forskydning i x-retningen. Denne variant fungerer desværre ikke, så vi kan kun forskyde screen'en vertikalt.

Vi indføjer følgende linier i programmet:

```
MoveScreen  = -162

    ...
    scrmove:                        ;* forskyd screen D0 mod højre og
                                    D1 nedad.
    move.l  intbase,a6              ;Intuitions-basisadresse til A6
    move.l  screenhd,a0             ;screen-handle til A0
    clr.l   d0                      ;ingen horisontal forskydning
    jsr     MoveScreen(a6)          ;og forskyd
    rts                             ;færdig
```

Vi skriver det første fuldstændige program, der gennemløber følgende trin:

1. **Åbning af Intutitions-biblioteket.**
2. **Åbning af en screen.**
3. **Forskydning af screen'en i den retning, der angives af joystick'et i port 2.**
4. **Lukning af screen'en ved tryk på affyringsknappen.**
5. **Lukning af Intuitions-biblioteket.**
6. **Afslutning.**

Til afslutning anvender vi et RTS, hvorved programmet også kan startes af SEKA med et "j run".

Og her er så det samlede program inklusive underrutiner, så man kan overskue dem med et blik:

```
;** Demo-program til åbning og forskydning af en screen **

Movescreen   = -162
OpenScreen   = -198
CloseScreen  = -66
CloseLibrary = -414
Open Lib     = -408             ;åben library
ExecBase     = 4                ;EXEC-basisadresse

joy2         = $dff00c          ;joystick 2 data
fyr          = $bfe001          ;affyringsknap 2: bit 7

run:
    bsr     openint             ;åben Intutition
    bsr     scropen             ;åben screen
```

```
        move    joy2,d6             ;gem joystick grunddata

loop:
        tst.b   fyr                 ;test affyringsknap
        bpl     afslutning          ;nedtrykket: afslutning

        move    joy2,d0             ;grunddata til D0
        sub     d6,d0               ;subtraher nye data

        cmp     #$0100,d0           ;opad?
        bne     noup                ;nej
        move.l  #-1,d1              ;dy=-1
        bsr     scrmove             ;skub opad
        bra     loop

noup:
        cmp     #$0001,d0           ;nedad?
        bne     loop                ;nej
        move.l  #1,d1               ;dy=1
        bsr     scrmove             ;skub nedad
        bra     loop

afslutning:
        bsr     scrclose            ;luk screen
        bsr     closeint            ;luk Intutition
        rts                         ;færdig!

openint:                            ;* initialiser og åben system
        move.l  ExecBase,a6         ;EXEC-basisadresse
        lea     IntName,a1          ;navn på Intuitionsbibliotek
        jsr     OpenLib(a6)         ;åben Intuition
        move.l  d0,intbase          ;gem Intuitions-basisadresse
        rts

closeint:                           ;* luk intuition
        move.l  execbase,a6         ;EXEC-basisadresse til A6
        move.l  intbase,a1          ;Intuitions-basisadresse til A1
        jsr     CloseLibrary(a6)    ;luk Intuition
        rts                         ;færdig

scropen:                            ;* åben screen
        move.l  intbase,a6          ;Intuitions-basisadresse til A6
        lea     screen_defs,a0      ;pointer på tabel
        jsr     openscreen(a6)      ;og åben
```

```
        move.l  d0,screenhd         ;gem screen-handle
        rts                         ;returner til hovedprogram

scrclose:                           ;luk screen
        move.l  intbase,a6          ;Intuitions-basisadresse til A6
        move.l  screenhd,a0         ;screen-handel til A0
        jsr     CloseScreen(a6)     ;luk screen
        rts                         ;færdig

scrmove:                            ;* forskyd screen D0 til højre og
                                    D1 nedad
        move.l  intbase,a6          ;Intuitions-basisadresse til A6
        move.l  screenhd,a0         ;screen-handle til A0
        clr.l   d0                  ;ingen horisontal forskydning
        jsr     MoveScreen(a6)      ;og forskyd
        rts                         ;færdig

even
screen_defs:                     ;* her begynder screen-tabellen
x_pos:          dc.w    0        ;x-position
y_pos:          dc.w    0        ;y-position
width:          dc.w    320      ;bredde
heigth:         dc.w    200      ;højde
depth:          dc.w    2        ;antal bitplanes
detail_pen:     dc.b    1        ;tekst-farve = hvid
block_pen:      dc.b    3        ;baggrundsfarve = rød
view_modes:     dc.w    2        ;repræsentationsmodus
screen_type:    dc.w    15       ;screen-type: customscreen
font:           dc.l    0        ;tegnsæt: standard
title:          dc.l    titel    ;pointer på titeltekst
gadgets:        dc.l    0        ;ingen gadgets
bitmap:         dc.l    0        ;ingen bitmap
intbase:        dc.l    0        ;basisadresse for Intuition
screenhd:       dc.l    0        ;screen-handle
IntName:        dc.b 'intuition.library',0
even
titel:          dc.b 'Vor screen',0    ;screen-overskrift
even
```

Man ser af dette eksempel, hvor nemt scrolling lader sig realisere, hvad der er særligt yndet ved spil.

En anden slags leg kan opnås via funktionen DisplayBeep, der har offset -96 og som parameter får overgivet screen-pointeren fra "screenhd" i A0. Funk-

tionen lader skærmen give et kort orange glimt. En enventuel anden synlig skærm forbliver uberørt heraf. Beep-funktionen udløses på følgende måde:

```
DisplayBeep = -96

        ...
        move.l intbase,a6         ;Intuitions-basisadresse til A6
        move.l screenhd,a0        ;screen-pointer til A0
        jsr    Displaybeep(a6)    ;lad skærmen glimte
```

Hvis man overgiver et nul i A0 i stedet for screen-pointeren, så blinker hele skærmbilledet.

Udmærket, så har vi vor egen skærm, som vi endda kan forskyde op og ned. Hvad nytte er det til, når vi ikke kan vise noget på den? Lad os derfor åbne et vindue på denne screen.

## 7.2. Åbning af vinduer

Som set i kapitlet om programforberedelse, er det ret simpelt at åbne et vindue med DOS-biblioteket. Metoden svigter desværre, når vi vil arbejde med vor egen skærm. Derfor må en anden metode læres, så vi kan åbne vilkårlige vinduer på vilkårlige screens.

Hertil har Intuition en funktion, der betegnende nok, kaldes OpenWindow. Den har offset -204, og kræver kun n parameter, nemlig, ligesom Open-Screen-funktionen, en pointer på en vindues-definitionstabel. Denne pointer overgives i register A0.

Tabellen ligner dem til screen-definitioner. Også her indføres først 4 ord, hvori angives x- og y-position samt bredde og højde på det vindue, der skal åbnes. Et eksempel herpå:

```
even
window_defs:
        dc.w    10    ;x-position
        dc.w    20    ;y-position
        dc.w   300    ;bredde
        dc.w   150    ;højde
```

Derefter kommer de to bytes, der definerer skrift- og baggrundsfarve:

```
        dc.b     1    ;skriftfarve hvid
        dc.b     3    ;på rød baggrund
```

Det lange ord, der nu følger, indeholder i sine bits de såkaldte IDCMP-flag, hvormed det bestemmes, ved hvilken hændelse Intuition skal overbringe programmet en melding. De enkelte bits har følgende betydning (værdi og navn på bit'en i parentes).

| Bit | Værdi | Navn | Betydning: melding ved |
|---|---|---|---|
| 0 | $000001 | SIZEVERIFY | |
| 1 | $000002 | NEWSIZE | Forandring af vinduesstørrelse |
| 2 | $000004 | REFRESHWINDOW | |
| 3 | $000008 | MOUSEBUTTONS | Tryk på en musetast |
| 4 | $000010 | MOUSEMOVE | Bevægelse af musen |
| 5 | $000020 | GADGETDOWN | Udvalg af en speciel gadgdet |
| 6 | $000040 | GADGETUP | som ovenfor |
| 7 | $000080 | REQSET | |
| 8 | $000100 | MENUPICK | Valg af et menupunkt |
| 9 | $000200 | CLOSEWINDOW | Lukning af vinduet |
| 10 | $000400 | RAWKEY | Tryk på en tast |
| 11 | $000800 | REQVERIFY | |
| 12 | $001000 | REQCLEAR | |
| 13 | $002000 | MENUVERIFY | |
| 14 | $004000 | NEWPREFS | Ændring af preferencerne |
| 15 | $008000 | DISKINSERTED | Indlægning af en diskette |
| 16 | $010000 | DISKREMOVED | Udtagning af en diskette |
| 17 | $020000 | WBENCHMESSAGE | |
| 18 | $040000 | ACTIVEWINDOW | Aktivering af vindue |
| 19 | $080000 | INACTIVEWINDOW | Inaktivering af vindue |
| 20 | $100000 | DELTAMAOVE | Meld musebevægelse relativt |

Det nye vindue skal kun reagere på aktivering af lukkesymbolet, så vi skriver:

```
dc.l    $200    ;IDCMP-flag: CLOSEWINDOW
```

Derefter kommer et langt ord, hvori bits'ene bestemmer typen af vinduet. Hermed kan det opbygges helt vilkårligt, i modsætning til de vinduer, der åbnes med DOS-funktionen. Bits'ene betyder

| Bit | Værdi | Navn | Betydning: melding ved |
|---|---|---|---|
| 0 | $0000001 | WINDOWSIZING | Vinduesstørrelse foranderlig |
| 1 | $0000002 | WINDOWDRAG | Vindue forskydeligt |
| 2 | $0000004 | WINDOWDEPTH | Vinduesoverlejring mulig |
| 3 | $0000008 | WINDOWCLOSE | Vindue-lukningssymbol |
| 4 | $0000010 | SIZEBRIGHT | |
| 5 | $0000020 | SIZEBOTTOM | |
| 6 | $0000040 | SIMPLE REFRESH | Nye tegn manuelt |
| 7 | $0000080 | SUPER BITMAP | Lager hele vinduesindhold |
| 8 | $0000100 | BACKDROP | Vindue bagud |
| 9 | $0000200 | REPORTMOUSE | Meld musekoordinater |
| 10 | $0000400 | GIMMEZEROZERO | Adskillelse af lister og vindue |
| 11 | $0000800 | BORDERLESS | Vinduer uden kanter |
| 12 | $0001000 | ACTIVATE | Vindue aktivt |
| 13 | $0002000 | WINDOWACTIVE | |
| 14 | $0004000 | INREQUEST | |
| 15 | $0008000 | MENUSTATE | |
| 16 | $0010000 | RMBTRAP | Højre musetast: ingen menu |
| 17 | $0020000 | NOCAREREFRESH | Ingen fornyelsesmelding |
| 24 | $1000000 | WINDOWREFRESH | |
| 25 | $2000000 | WBENCHWINDOW | |

Refresh betyder kun, at vinduesindholdet skal opbygges på ny, f.eks. efter en forstørrelse af vinduet. Hvis ingen af refresh-bit'ene sættes, så betyder det Smart-Refresh-modus, hvor Intuition selv sørger for fornyelsen af vinduet. Det er den letteste, og derfor den mest almindelige måde.

Hvis vi i vindueseksemplet vælger værdien $100F som type, så bliver vinduet straks aktivt ved åbningen, og bærer alle system-gadgets:

```
        dc.l    $100f               ;ACTIVATE og alle gadgets
```

Det næste lange ord i listen er beregnet på, at man vil bringe egne gadgets ind i vinduet. Så kommer der en pointer på disse gadgets' struktur. Men da vi *ikke* vil det, sætter vi et nul.

```
        dc.l    0                   ;FirstGadget: ingen egne gadgets
```

Det næste lange ord er til en pointer, på en grafik-struktur for egne symboler til hentning af menupunkter. Vi sætter også det på nul, for således at indstille standard-tegnet:

```
dc.l      0                ;check-mærke: standard
```

Den næste indførelse er en pointer på teksten, der skal tjene som vindues-overskrift. Også denne tekst skal afsluttes med en nul-byte.

```
dc.l      windowname       ;pointer på vinduestitel
```

Nu følger, som et langt ord, pointeren på screen-strukturen, som vi har fået med OpenScreen-kaldet. Den simpleste metode til at indføre det, er at indsætte bufferen til at gemme pointeren:

```
screenhd: dc.l  0          ;screen-pointer
```

Næste lange ord er pointer på en bitmap, hvis man vil anlægge sin egen til vinduet. Da vi heller ikke vil det, sætter vi også her et nul:

```
dc.l      0                ;ingen egen bitmap
```

Nu følger fire ord, hvori mindstebredden og -højden, samt den maksimale bredde og højde af vinduet angives:

```
dc.w    150                ;mindstebredde
dc.w     50                ;mindstehøjde
dc.w    320                ;maksimale bredde
dc.w    200                ;maksimale højde
```

Endelig angiver det sidste ord på listen den screen-type, hvori vinduet skal ligge. Vi indfører et 15 igen, så vor screen anlægges som en customscreen:

```
dc.w     15                ;screen-type: custom-screen
```

Her er hele listen:

```
even
window_defs:
        dc.w    10         ;x-position
        dc.w    20         ;y-position
        dc.w   300         ;bredde
        dc.w   150         ;højde
        dc.b     1         ;skriftfarve hvid
```

```
        dc.b      3          ;på rød baggrund
        dc.l  $200           ;IDCMP-flag: CLOSEWINDOW
        dc.l $100f           ;ACTIVATE og alle gadgets
        dc.l      0          ;FirstGadget: ingen egne gadgets
        dc.l      0          ;CheckMark: standard
        dc.l windowname      ;pointer på vindues-titel
screenhd:  dc.l  0           ;screen-pointer
        dc.l      0          ;intet eget bitmap
        dc.w   150           ;mindstebredde
        dc.w    50           ;mindstehøjde
        dc.w   320           ;maksimale bredde
        dc.w   200           ;maksimale højde
        dc.w    15           ;screen-type: custom-screen
```

og naturligvis også vindues-titlen:

```
windowname:  dc.b  'Vort vindue',0
even
```

Disse linier indføjes i programmet, som listet ovenfor. Hertil kommer to underrutiner, til at åbne og lukke vinduet

```
OpenWindow   = -204
CloseWindow  = -72

        ...
windowopen:
        move.l intbase,a6         ;Intutitions-basis til A6
        lea    windowdef,a0       ;pointer på vinduesdefinition
        jsr    openwindow(a6)     ;åben vindue
        move.l d0,windowhd        ;gem vindues-handle
        rts

windclose:
        move.l intbase,a6         ;Intuitions-basis til A6
        move.l windowhd,a0        ;vindues-handle
        jsr    closewindow(a6)    ;luk vindue
        rts

        ...
windowhd:  dc.l  0                ;vindues-handle
```

Nu kan man indføje et "bsr windopen" bag "bsr scropen", og et "bsr windclose" før "bsr scrclose" i programeksemplet. Når man starter programmet, forskydes det nye vindue sammen med screen'en, da det jo ligger indenfor samme. Prøver man at skubbe vinduet ud af screen'en med musen, vil man opdage, at det ikke er muligt.

I eksempelet bærer vinduet i øverste venstre hjørne lukke-gadget'en, hvormed man sædvanligvis kan lukke et vindue ved at "klikke den ind". Prøv: det har ingen effekt.

Brugen af gadgets, så vel som andre hændelser, skal programmeres, da Intuition ikke ved, hvilken handling, der skal følge efter. I næste kapitel ses på behandlingen af tildragelser.

## 7.3. Meldinger og aflæsninger: Requester

Læseren kender meldingen fra Amiga'en: "Please insert xxx in unit 0", hvis man kun har en diskettestation. Denne melding udlæses i et ekstra vindue, der yderligere har to felter, man kan klikke ind på. Den form for melding med valgmulighed, kalder man Requester.

*Nu en simpel måde til at programmere requestere. Hertil behøver vi et vindue, som requesteren kan opstå i; men sådan et vindue har vi allerede åbnet i programeksemplet.*

For at angive en requester, benytter vi Intutitions-funktionen AutoRequest med offset -348, der automatisk overvåger oprettelsen og styringen. Funktionen kræver følgende parametre:

**I A0** Pointeren på den vindues-struktur, som vi har lagret i "windowhd"

**I A1** En pointer på strukturen af den tekst, der skal stå over valgknapperne.

**I A2** Som ovenfor for den venstre knaps tekst.

**I A3** Som ovenfor for den højre knap

**I D0** IDCMP-flagene, der angiver, hvilken hændelse et tryk på den venstre knap skal svare til.

**I D1** Som ovenfor for den højre knap.

**I D2** Bredden af hele requesteren.

**I D3** Højden af requesteren.

Vi indføjer følgende linier i vort program:

```
AutoRequest = -348

    ...
request:
        move.l  windowhd,a0         ;pointer på vinduesstruktur
        lea     btext,a1
        lea     ltext,a2            ;pointer på tekststrukturer
        lea     rtext,a3
        move.l  #0,d0               ;kun til venstre ved at klikke ind
        move.l  #0,d1               ;kun til højre ved at klikke ind
        move.l  #180,d2             ;bredde og
        move.l  #80,d3              ;højde af requesteren
        move.l  intbase,a6          ;Intuitions-basis
        jsr     autorequest(a6)     ;vis requester
        rts
```

De flag, der blev overgivet i D1 og D2, byder på interessante muligheder. Således er også de systemmeldinger, der opfordrer til at indlægge en bestemt diskette, forsynet med DISKINSERTED-flaget, hvad der gør indlægningen af en diskette til den samme hændelse, som at klikke sig ind på "retry".

Det nye er begrebet "tekst-struktur", som er dukket op tre gange. Det drejer sig om en liste, hvori er indført nogle oplysninger om den pågældende tekst.

Listen begynder med to bytes, hvori farverne defineres. Den første byte står for tekstfarven, den anden for baggrundsfarven, som dog ikke har nogen betydning her.

```
btext:
        dc.b  2         ;tekstfarve sort
        dc.b  0         ;baggrundsfarve
```

Derefter følger endnu en byte, der angiver tegn-modus. Tilføjer man et 0, så udlæses teksten normalt, et 4 bevirker inverteret visning af teksten.

```
        dc.b  0         ;normal tekstafbildning
```

Dernæst følger to ord på listen. Derfor skal adressen først gøres lige, idet man enten tilføjer en byte, eller anvender "even"-direktivet. De ord, der følger, angiver tekstens x- og y-position i forhold til øvre venstre hjørne af den "kasse", hvori den skal skrives. Kassen er for requester- overskriften requester-vinduet, ved valg-teksterne er det valgknappen.

```
        dc.w  10        ;x-afstand
        dc.w   5        ;y-afstand til øvre venstre hjørne
```

Så følger en pointer på det tegnsæt, der skal anvendes. Sæt et nul, for at vælge standard-tegnsættet.

```
        dc.l  0             ;standard tegnsæt
```

Derefter skal vi overgive adressen på den tekst, der skal udlæses. Teksten skal afsluttes med en nulbyte.

```
        dc.l  text          ;pointer på text
```

Som afslutning følger et langt ord, der enten peger på en tekst, eller er nul, hvis der ikke er mere tekst.

```
        dc.l  0             ;ikke mere tekst
```

Her de tre tekststrukturer til eksemplet:

```
btext:                      ;overskriftens tekststruktur
        dc.b  0,1           ;farver
        dc.b  0             ;modus
even
        dc.w  10,10         ;tekstposition
        dc.l   0            ;standardtyper
        dc.l  bodytxt       ;pointer på tekst
        dc.l  0             ;ikke mere tekst

bodytxt: dc.b "Requester-tekst",0
even

ltext:                      ;venstre knaps tekststruktur
        dc.b  0,1           ;farver
        dc.b  0             ;modus
even
        dc.w  5,3           ;tekstposition
        dc.l  0             ;standard typesæt
        dc.l  lefttext      ;pointer på tekst
        dc.l  0             ;ikke mere tekst

lefttext: dc.b "venstre",0
even

rtext:
        dc.b  0,1           ;farver
        dc.b  0             ;modus
```

```
even
        dc.w    5,3          ;tekst-position
        dc.l    0            ;standard typesæt
        dc.l    righttext    ;pointer på tekst
        dc.l    0            ;ikke mere tekst

righttext: dc.b "hoejre",0
even
```

Efter at have kaldt denne requester, får man i D0 information om, hvilken af de to knopper, der er trykket på, resp. i hvilken knop hændelsen har fundet sted. Hvis D0 er nul, så var det den højre knop, er den 1, var det venstre.

## 7.4. Behandling af hændelser

Forudsæt at vi har åbnet et vindue, der bærer lukkesymbolet, og vil reagere på aktiveringen af symbolet. Dertil behøves et signal fra Intuition, som angiver, at der er indtrådt en hændelse. Signalet kaldes "message", på engelsk meddelelse.

Efter hviken hændelse meldingen skal komme, angives i vinduets IDCMP-flag: ved at sætte bittene for WINDOWCLOSE bliver hændelsen "lukke-symbol aktiveret" gjort til en mulig årsag til en melding.

For at bevare denne melding findes en EXEC-funktion, nemlig funktionen GetsMsg med offset -372, der får adressen på hændelseskilden som parameter. I vort tilfælde er kilden den såkaldte user-port, der ikke har noget at gøre med user-porten på en af de gamle Commodore-computere.

User-porten udgør en tabel, hvori hændelserne og deres randbetingelser som museposition og klokkeslet indføres af Intuition.

Hvordan finder man denne user-port? Til den ende ser vi på pointeren på vinduesstrukturen, som vi får fra OpenWindow-funktionen og har lagret i "windowhd".

Pointeren peger, på det pågældende vindues struktur, som består af indførelser, der til dels er kopier af parametrene fra vor vindues-definitionstabel. Vi vil ikke gå nærmere ind på disse, da de ikke er interessante. Vigtigst er pointeren på den søgte user-port, som indeholdes i vinduesstrukturen.

Det er det lange ord, der begynder i den 86. byte af strukturen. Vi får fat i det lange ord, ved at programmere på følgende måde:

```
move.l windowhd,a0 ;pointer på strukturen i A0
move.l 86(a0),a0    ;user-port-pointer til A0
```

Med pointeren i A0 kan vi kalde GetMsg-funktionen, idet vi indføjer følgende linier:

```
GetMsg = -372

...
move.l windowhd,a0 ;pointer på struktur i A0
move.l 86(a0),a0    ;user-port-pointer til A0
move.l ExecBase,a6 ;EXEC-basisadresse til A6
jsr    GetMsg(a6)  ;hent message
```

Værdien returnerer straks, og giver en returværdi i D0. Denne værdi er igen en pointer, der peger på Intuitions-message-strukturen. Hvis der returneres et 0 i D0, har der ikke været nogen hændelse.

Det lange ord, der ligger fra den 20. byte af strukturen, indeholder information om, hvilken hændelse, der er indtrådt. Bit'ene i det lange ord har samme betydning som i IDCPM-flagene, der er beskrevt under vinduesåbning.

Ovenstående linier skal tilføjes efter "loop" tillige med følgende fire linier:

```
move.l d0,a0        ;message-pointer til A0
move.l 20(a0),d6    ;gem hændelse i D6
tst.l  d0           ;er der sket en hændelse?
bne    afslutning   ;ja!
```

Når man starter programmet, kan man afslutte det ved at klikke lukkesymbolet ind. På den måde kan vi fastslå, om der har fundet en hændelse sted. I D6 kan man desuden fastslå, hvilken det var. I vort eksempel står tallet $00000200, der betyder, at lukkesymbolet blev klikket ind.

For at prøve, om det også går med andre hændelser, kan man sætte $10200 som IDCMP-flag i vindues-definitonstabellen i stedet for $200. Når man har assemblet, og startet denne version, tager man disketten ud af stationen: programmet afbrydes.

Det ekstra IDCMP-flag bevirker, at der foruden en "indklikning" af lukkesymbolet bliver meldt, at disketten er taget ud (DISKREMOVED). Hvilken af de to hændelser, der har fundet sted, kan man se i D6. Her står $200 hvis vinduet er lukket, og $10000 hvis disketten er taget ud.

# 7.5. Menu-programmering

Nu til en af Intuitions allermest interessante egenskaber: menu-programmering. Man kan gøre sit program bekvemt med menuer.

*Mulighederne for at afbilde menuer, er mangfoldige. Således kan man gøre menupunkterne ikke-valgbare, udlæse undermenuer og vælge arten af menuindførelser, om man vil udlæse tekster eller billeder. Mangfoldigheden af muligheder kræver også en del parametere.*

Der skal fremstilles en menu ved hjælp af Intuitions SetMenuStrip med offset -264. Funktionen behøver to parametre, nemlig en pointer på menustrukturen og pointeren på det vindue, hvori menuen skal fungere. Hvert vindue kan have sin egen menu, der bliver aktiv, når vinduet aktiveres.

Her er underrutinen til at sætte menuen:

```
SetMenuStrip = -264

    ...
    setmenu:                        ;* initialiser en menu
    move.l  intbase,a6              ;Intuitions-basis til A6
    move.l  windowhd,a0             ;pointer på vinduesstruktur
    lea     menu,a1                 ;pointer på menustruktur
    jsr     SetMenuStrip(a6)        ;funktionskald
    rts
```

Og naturligvis rutinen til at slette menuen:

```
ClearMenuStrip = -54

    ...
    move.l  intbase,a6              ;Intuitions-basis til A6
    move.l  windowhd,a0             ;pointer på vinduesstruktur
    jsr     ClearMenuStrip(a6)
    rts
```

Pointeren på vinduesstrukturen har vi allerede. Sådan en struktur skal opbygges for hver menu, dvs. for hver menutitel, der fremkommer, når man trykker på højre musetast.

Strukturen er en tabel, der skal have følgende opbygning:

Først kommer et langt ord, der peger på strukturen af den følgende menu. Hvis den aktuelle menu er den sidste, indføres et nul.

```
even
menu:
         dc.l   menu1            ;pointer på næste menu
```

Derefter følger to ord, der angiver menutitlens x- og y-position:

```
         dc.w   20               ;x-position
         dc.w    0               ;y-position
```

Og dernæst menutitlens bredde og højde i to ord, angivet som skærmbilledpunkter:

```
         dc.w   50               ;bredde
         dc.w   10               ;højde af menutitel
```

det næste ord indeholder en flag-bit, der bestemmer, om menuen er valgbar. En ikke valgbar menu kan genkendes på, at alle indførelser vises i grå farve resp. ret svagt. Hvis flag-bit'en, bit 0, er sat, så er menuen valgbar, ellers ikke.

```
         dc.w    1               ;menu valgbar
```

Nu følger et langt ord, nemlig pointeren på den tekst, der skal vises som menutitel. Pas på, at tekstlængden ikke er større, end tilladt af breddeangivelsen! Ellers sker uheldige ting.

```
         dc.l   menutext         ;pointer på titel-tekst
```

Og så endnu et langt ord, der udgør en pointer på det første felt i denne menu. Hver menufelt kræver desværre sin egen struktur.

```
         dc.l   menuitem01       ;pointer på det første menupunkt
```

Afslutningen af tabelen dannes af fire ord, der er reserveret til interne funktioner, men skal være til stede.

```
         dc.w   0,0,0,0          ;reserverede ord
```

Det var strukturen af første menu. Efter eksemplet ovenfor peger det første lange ord på den næste struktur, der har samme opbygning. Ved den sidste menu sættes pointeren lig nul.

Hvad vi mangler, er strukturen i menufelterne. Disse strukturtabeller har følgende opbygning:

# 7.5. Menu-programmering

Nu til en af Intuitions allermest interessante egenskaber: menu-programmering. Man kan gøre sit program bekvemt med menuer.

*Mulighederne for at afbilde menuer, er mangfoldige. Således kan man gøre menupunkterne ikke-valgbare, udlæse undermenuer og vælge arten af menuindførelser, om man vil udlæse tekster eller billeder. Mangfoldigheden af muligheder kræver også en del parametere.*

Der skal fremstilles en menu ved hjælp af Intuitions SetMenuStrip med offset -264. Funktionen behøver to parametre, nemlig en pointer på menustrukturen og pointeren på det vindue, hvori menuen skal fungere. Hvert vindue kan have sin egen menu, der bliver aktiv, når vinduet aktiveres.

Her er underrutinen til at sætte menuen:

```
SetMenuStrip  = -264

    ...
    setmenu:                        ;* initialiser en menu
    move.l  intbase,a6              ;Intuitions-basis til A6
    move.l  windowhd,a0             ;pointer på vinduesstruktur
    lea     menu,a1                 ;pointer på menustruktur
    jsr     SetMenuStrip(a6)        ;funktionskald
    rts
```

Og naturligvis rutinen til at slette menuen:

```
ClearMenuStrip  = -54

    ...
    move.l  intbase,a6              ;Intuitions-basis til A6
    move.l  windowhd,a0             ;pointer på vinduesstruktur
    jsr     ClearMenuStrip(a6)
    rts
```

Pointeren på vinduesstrukturen har vi allerede. Sådan en struktur skal opbygges for hver menu, dvs. for hver menutitel, der fremkommer, når man trykker på højre musetast.

Strukturen er en tabel, der skal have følgende opbygning:

Først kommer et langt ord, der peger på strukturen af den følgende menu. Hvis den aktuelle menu er den sidste, indføres et nul.

```
even
menu:
        dc.l    menu1           ;pointer på næste menu
```

Derefter følger to ord, der angiver menutitlens x- og y-position:

```
        dc.w    20              ;x-position
        dc.w     0              ;y-position
```

Og dernæst menutitlens bredde og højde i to ord, angivet som skærmbilledpunkter:

```
        dc.w    50              ;bredde
        dc.w    10              ;højde af menutitel
```

det næste ord indeholder en flag-bit, der bestemmer, om menuen er valgbar. En ikke valgbar menu kan genkendes på, at alle indførelser vises i grå farve resp. ret svagt. Hvis flag-bit'en, bit 0, er sat, så er menuen valgbar, ellers ikke.

```
        dc.w     1              ;menu valgbar
```

Nu følger et langt ord, nemlig pointeren på den tekst, der skal vises som menutitel. Pas på, at tekstlængden ikke er større, end tilladt af breddeangivelsen! Ellers sker uheldige ting.

```
        dc.l    menutext        ;pointer på titel-tekst
```

Og så endnu et langt ord, der udgør en pointer på det første felt i denne menu. Hver menufelt kræver desværre sin egen struktur.

```
        dc.l    menuitem01      ;pointer på det første menupunkt
```

Afslutningen af tabelen dannes af fire ord, der er reserveret til interne funktioner, men skal være til stede.

```
        dc.w    0,0,0,0         ;reserverede ord
```

Det var strukturen af første menu. Efter eksemplet ovenfor peger det første lange ord på den næste struktur, der har samme opbygning. Ved den sidste menu sættes pointeren lig nul.

Hvad vi mangler, er strukturen i menufelterne. Disse strukturtabeller har følgende opbygning:

Det begynder med en pointer på det næste menupunkt, der ved sidste felter sættes lig nul:

```
even
menuitem01:
    dc.l    menuitem02    ;pointer på næste menupunkt.
```

Derefter følger igen fire ord, hvori der indføres x- og y-position samt bredde og højde af den "box", hvori menuparametrene står. Disse mål bliver ikke direkte synlige; først når man vælger punktet med musens pointer og nedtrykt højre musetast, ser man boxens omrids. Hvordan man ser dem, bestemmes af flagene i det næste ord. Men først til position og mål for menupunkterne.

```
    dc.w     0    ;x-position for indførels
    dc.w     0    ;y-position
    dc.w    90    ;bredde i pixels
    dc.w    10    ;højde i pixels
```

Postionsangivelserne er relative i forhold til den nedrullede menus øvre venstre hjørne.

Det ord, der følger indeholder flag for angivelserne om det pågældende menupunkt. Visse variationerer mulige. Følgende bit-flag er indeholdt i ordet:

| Bit | Værdi | Navn | Betydning når sat |
|---|---|---|---|
| 0 | $0001 | CHECKIT | Punktet bliver taget ned ved valg |
| 1 | $0002 | ITEMTEXT | Tekst-menupunkt |
| 2 | $0004 | COMMSEQ | Valg er også muligt med en tast |
| 3 | $0008 | MENUTOGGLE | Nedtagning kobles til og fra |
| 4 | $0010 | ITEMENABLED | Menupunkt valgbart |
| 6 | $0040 | HIGHCOMP | Punkt inverteres ved valg |
| 7 | $0080 | HIGHBOX | Punkt indrammes ved valg |
| 8 | $0100 | CHECKED | Punktet er hentet ned |

Betydningen af disse punkter er følgende:

**CKECKIT** Hvis denne bit er sat, så bliver der automatisk sat en "krog" eller et selvdefineret tegn før teksten, når punktet er blevet valgt. Teksten bør begynde med to tomme mellemrum.

**ITEMTEXT** Menupunktet er en normal tekst, hvis denne bit er sat. Ellers udlæses en tegning.

**COMMSEQ** Ved at sætte bit'en og angive et tegn (se senere) kan dette menupunkt også vælges ved at aktivere den højre Amiga-tast og den angivne tast. Tegnet vises også i menuen med Amiga-symbolet, hvortil man bør sørge for plads.

**MENUTOGGLE** Hvis bit'en er sat og nedhentning tilladt (bit 0), så slettes "krogen" automatisk anden gang punktet vælges, sættes igen næste gang osv.

**ITEMENABLED** Ved sletning af bit'en bliver menupunktet ikke-valgbart

**HIGHCOMP** Når bit'en er sat, bliver den definerede box omkring menupunktet inverteret ved berøring med musens pointer.

**HIGHBOX** Med denne modus bliver boxen indrammet, hvis den berøres.

De to sidste bits angiver den modus, hvori et bestemt menupunkt skal fremkomme. Følgende kombinationer er tillige mulige:

**HIGHIMAGE** Hvis begge bits er slettet, bliver der udlæst en selvdefineret tegning ved berøring af boxen.

**HIGHNONE** Når begge bits er sat, sker intet ved berøring af kassen.

**CHECKED** Denne bit sættes både af programmet og af Intuition. Den angiver, om der skal stå en "krog" foran menuteksten. Man kan altså fastslå fra sit program, om punktet blev hentet ned, ved at teste bit 8 i dette ord. Hvis den er sat, så er det hentet ned. Ellers kan man sætte den fra programmet, og punktet vil fremkomme hentet ned.

Vi vælger disse modi CHECKIT, ITEMTEXT, COMMSEQ, MENUTOGGLE, ITEMENABLED og HIGHBOX til vort eksempel:

```
dc.w    %10011111    ;modus-flag
```

Men nu tilbage til strukturen af menupunkterne. Efter flag-ordet følger et langt ord, hvori kan vælges, om dette menupunkt udelukker et andet. Vi sætter det på nul:

```
        dc.l    0              ;ingen tilslutning
```

Nu kommer pointeren på strukturen af den tekst, der skal vises. Hvis ITEMTEXT-bit'en ikke er sat, skal pointeren pege på tegningens struktur. Hvis der ikke skal vises noget, så kan man sætte pointeren på nul. Vi anvender en test og skriver derfor:

```
        dc.l    menu01text     ;pointer på menutekst-struktur
```

Det følgende lange ord har kun betydning, når HIGHIMAGE-flaget er sat. Så peger langordet på den tekst eller tegning, der skal vises ved berøring af menupunktet. Ellers ignoreres det lange ord, hvorfor vi sætter det på nul:

```
        dc.l    0              ;ingen tegning ved berøring
```

Den næste parameter er en byte, til angivelse af det tastaturtegn, der kan vælge det aktuelle menupunkt, når den nedtrykkes sammen med Amiga-tasten. Det gælder kun, når COMMSEQ-bit'en er sat. Vi skriver et tegn ind:

```
        dc.b    'A'            ;kan også vælges med Amiga/'A'
```

Da der nu atter kommer et langt ord, skal adressen først rettes ud med "even". Derefter kommer det lange ord, der peger på menustrukturen i en eventuel undermenu. Disse undermenuer bliver automatisk vist, når man berører menupunktet med musens pointer. Desværre er det ikke muligt med en videre underafdeling, så det lange ord kan ignoreres ved en undermenu.

*Vil man ikke oprette nogen undermenu, skal der tilføjes et nul:*

```
    even
        dc.l    0              ;ingen undermenu
```

Det sidste lange ord bliver der skrevet i, af Intuition, når man vælger flere menupunkter. I så fald står her menunummeret på den næste menustruktur, der er valgt.

```
        dc.l    0              ;forberedt
```

Endnu mangler dette punkt tekststrukturen for den tekst, der skal vises. Den tillader nogle flere finesser i udformningen af menuen.

Her er hele strukturen af et menueksempel. Der vises to menuer, hver med to underpunkter. Det andet underpunkt i den venstre menu har tillige en undermenu med to indførelser. Det anbefales at indtaste denne programdel,

da den er god at eksperimentere med. Desuden skal vi lære af dette eksempel, hvordan man udnytter de menupunkter, man har klikket sig ind på.

```
;** komplet menustruktur for menueksempel **

menu:
    dc.l  menu1             ;ingen følgende menu
    dc.w  10,30             ;x/y
    dc.w  50,10             ;bredde/højde
    dc.w  1                 ;MenuEnabled
    dc.l  menunavn          ;menutitel
    dc.l  menuitem01        ;menuparameter
    dc.w  0,0,0,0
menunavn:
    dc.b  ”Menu 1”,0        ;første menu
even

menu1:
    dc.l  0                 ;ikke flere menuer
    dc.w  80,0              ;se ovenfor
    dc.w  50,10
    dc.w  1
    dc.l  menunavn1
    dc.l  menuitem11
    dc.w  0,0,0,0
menunavn1:
    dc.b  ”Menu 2”,0        ;andet menunavn
even

menuitem01:                 ;første menupunkt
    dc.l  menuitem02        ;pointer på næste parameter
    dc.w  0,0               ;x,y
    dc.w  130,12            ;bredde,højde
    dc.w  $9f               ;flag
    dc.l  0                 ;Exclude
    dc.l  text01            ;pointer på tekststruktur
    dc.l  0                 ;SelectFill
    dc.b  ”1”               ;Command
even
    dc.l  0                 ;subitem: intet
    dc.w  0                 ;NextSelect: nej
text01:
    dc.b  0,1               ;farver
    dc.b  0                 ;modus: skriv oven i
```

```
even
    dc.w 5,3            ;x/y-position
    dc.l  0             ;standard-tegnsæt
    dc.l  text01txt     ;pointer på tekst
    dc.l  0             ;ikke mere tekst

text01txt:
    dc.b  " Punkt 0.1",0
even

menuitem02:             ;andet menupunkt
    dc.l  0
    dc.w 0,10
    dc.w 130,12
    dc.w $57
    dc.l  0
    dc.l  text02
    dc.l  0
    dc.b  "2"           ;aktivering med Amiga/'2'
even
    dc.l  0
    dc.w 0

text02:
    dc.b  0,1
    dc.b  0

even
    dc.w 5,3
    dc.l  0
    dc.l  text02txt
    dc.l  0

text02txt:
    dc.b  " Punkt 0.2",0
even

menuitem11:             ;første menupunkt i anden menu
    dc.l  menuitem12    ;pointer på andet menupunkt
    dc.w 0,0
    dc.w 90,12
    dc.w $52
    dc.l  0
    dc.l  text11
```

```
    dc.l  0
    dc.b  0
even
    dc.l  0
    dc.w  0

text11:
    dc.b  0,1
    dc.b  0
even
    dc.w  5,3
    dc.l  0
    dc.l  text11txt
    dc.l  0
text11txt:
    dc.b  "Punkt 1.1",0
even

menuitem12:                 ;andet menupunkt i anden menu
    dc.l  0                 ;ikke flere punkter
    dc.w  0,10
    dc.w  90,12
    dc.w  $92
    dc.l  0
    dc.l  text12
    dc.l  0
    dc.b  0
even
    dc.l  submenu0          ;pointer på undermenu
    dc.w  0

text12:
    dc.b  0,1
    dc.b  0
even
    dc.w  5,3
    dc.l  0
    dc.l  text12txt
    dc.l  0
text12txt:
    dc.b  "Punkt 1.2",0
even

submenu0:                   ;undermenu, første punkt
```

```
    dc.l  submenu1              ;pointer på næste punkt
    dc.w 80,5
    dc.w 90,12
    dc.w $52
    dc.l  0
    dc.l  texts0
    dc.l  0
    dc.b  0
even
    dc.l  0
    dc.w 0

texts0:
    dc.b  0,1
    dc.b  0
even
    dc.w 5,3
    dc.l  0,texts0txt,0
texts0txt:
    dc.b  "U-punkt 1",0
even

submenu1:                   ;undermenu, andet punkt
    dc.l  0
    dc.w 80,15
    dc.w 90,12
    dc.w $52
    dc.l  0
    dc.l  texts1
    dc.l  0
    dc.b  0
even
    dc.l  0
    dc.w 0

tests1:
    dc.b  0,1
    dc.b  0
even
    dc.w 5,3
    dc.l  0
    dc.l  texts1txt
    dc.l  0
texts1txt:
```

```
        dc.b  "U-punkt 2",0
    even
```

De enkelte menupunkter i dette eksempel har følgende egenskaber, der er indstillet af flagene:

### Menu 1

Det første punkt, "Punkt 0.1" kan også vælges med tasterne Amiga højre og "1". Punktet bliver skiftevis hentet ned og ikke hentet ned, hvorved det bliver let at teste tast-funktionerne. Hvis "krogen" ikke er sat, og man trykker på disse to taster, så fremkommer krogen og omvendt. Desuden bliver dette punkts box indrammet, når musens pointer rør ved det.

Det andet punkt, "punkt 0.2" kan vælges med tast "2" (og Amiga til højre). Punktet hentes ned ved det første valg, men krogen lader sig ikke slette. Dette punkts box bliver inverteret ved berøring.

### Menu 2

Disse to punkter kan ikke vælges med taster. Det øverste punkts box bliver inverteret ved berøring, det nederste indrammet. Ved berøring af det andet punkt, "Punkt 1.2" vises en undermenu, som har to felter.

*Man kan nemt eksperimentere med strukturen ved at ændre nogle flag. Som det ses, er menuprogrammering ikke så kompliceret. Det giver omfattende skriveri, men er til gengæld meget alsidigt.*

Prøv at skrive eget program med menuer. Straks dukker det spørgsmål op, hvordan programmet kan fastslå, hvilket punkt i hvilken menu, der blev aktiveret.

Muligheden for at konstatere menutilstanden er allerede beskrevet: testen af CKECKED-bit'en i et menupunkts flag-ord. Hvis den er sat, har brugeren klikket ind på menupunktet med musen.

Det fungerer kun, hvis det er tilladt at hente det testede menupunkt ned. Men bortset fra muligheden af at forsyne alle menupunkter med "kroge", er det møjsommelig programmering at teste alle flag-bits.

En metode til at få meddelelser fra intuition om en hændelse, har vi lært. Typen af hændelse, der meldes, er lagt i register D6, og det kan ses, hvad der er sket.

Når vi sætter bit 8, MENUPICK i vinduesdefinitionens IDCMP-flaglangord, bliver valget af et menupunkt meldt af GetMsg-funktionen. Vi skriver følgende linier (hvis de ikke allerede står der) i løkken:

```
loop:
        move.l  execbase,a6       ;EXEC-basisadresse til A6
        move.l  windowhd,a0       ;vinduesstruktur-pointer
        move.l  86(a0),a0         ;user-port-pointer til A0
        jsr     GetMsg(a6)        ;hent message
        tst.l   d0                ;var der nogen?
        beq     loop              ;ingen hændelse

        move.l  d0,a0             ;message-pointer til A0
        move.l  $14(a0),d6        ;hændelse til D6
```

Når programmet er nået til dette sted, er der sket en hændelse, hvis flag vi har liggende i D6. Vi kan anvende hændelsen, idet vi tester flag-bits'ene med CMP eller BTST og, hvis de er sat, udløser de tilsvarende funktioner. Således kan vi med følgende linier:

```
        cmp     #$200,d6          ;WINDOWCLOSE?
        beq     afslutning        ;ja: programafslutning
```

afslutte programmet ved at lukke vinduet.

Hvis brugeren nu har valgt et menupunkt, meldes det med værdien $100 i D6. Vi skal fastslå, hvilket menupunkt det var.

Informationen findes i et ord, der står lige bagved det lange ord med hændelsesflaget i message-strukturen. Vi skriver:

```
        move    $18(a0),d7
```

og får koden for menupunktet der er klikket til i D7. Har man kun trykket den højre Amiga-tast og sluppet den, uden at vælge et menupunkt, finder vi værdien $FFFF her. Ordet indeholder ellers ikke blot n, men hele tre informationer:

- **I hvilken menu blev der valgt?**
- **Hvilket menupunkt?**
- **Hvilken undermenu?**

Informationerne er fordelt på tre bit-grupper, på følgende bits:

| | |
|---|---|
| **Bits 0 - 4** | **Menutitel-nummer** |
| **Bits 5 - 10** | **Menupunkt-nummer** |
| **Bits 11 - 15** | **Undermenupunkt-nummer** |

Alle numrene begynder ved 0. Dvs. det første punkt i den første menu har numrene 0 og 0.

For at afprøve det kan man tilføje følgende linier:

```
         move    d7,d6      ;bring kode til D6
         lsr     #8,d7      ;forskyd 11 gange mod højre
         lsr     #3,d7      ;undermenupunktet er nu i D7
         clr.l   d5
         roxr    #1,d6      ;bit 0 i x-flaget
         roxl    #1,d5      ;menunummer nu til D5
         and.l   #$7f,d6    ;udskil underste bits
         cmp     #$7f,d6    ;intet menupunkt?
         beq     loop       ;nej: videre
         lsr     #4,d6      ;ellers menupunkt til D6
afslutning:
```

Ved en testkørsel med SEKA konstateres, om vurderingen fungerer rigtigt, da man kan se resultatet direkte i de anførte registre, efter programmets afslutning.

Vil man skrive et program med 4 menuer, hver med 10 menupunkter, er ovenstående skrivemåde for besværlig til 44 tabeller. Her et program, der fremstiller de nødvendige strukturtabeller.

Den frembragte menustruktur, er meget enkelt opbygget, dvs. uden specielle valgmodi eller undermenuer. Hvis man alligevel vil have ekstraudstyr, kan man stadig anvende programmet, men skal bagefter, tilføre de ønskede flag eller pointere med MOVE-kommandoer.

De oplysninger, som programmet kræver, er blot navnene på menuerne og punkterne, der skal stå i menuen. Adresserne på teksterne bliver lagt ind i en tabel, som har følgende opbygning:

```
dc.l   menutitel 1
dc.l   punkt1, punkt2, punkt3,...,0
dc.l   menutitel 2
dc.l   punkt1, punkt2, punkt3,...,0
dc.l   menutitel 3 eller 0
```

Programmet er konstrueret med indtil 4 menuer ved siden af hinanden (med normal skærmbilledopløsning), hvad ofte er tilstrækkeligt. Tabellen ovenfor ender med, at der i stedet for pointeren, på den næste menutitel, tilføjes et nul. Man ser, hvor nemt det er!

Programmet indføjes i det samlede program, lige bagved label'en "setmenu". Således bliver menustrukturen opbygget og samtidig initialiseret blot ved, at kalde "bsr setmenu". Resten af programmet ændrer sig ikke, det bliver blot kortere.

Her er programdelen som "setmenu"-rutine:

```
setmenu:                          ;*initialiser menustruktur
    lea      mentab,a0            ;pointer på tekstpointer i A0
    lea      menu,a1              ;pointer på menufelt i A1
    move     #10,d1               ;horisontal menuposition=10

menuloop:
    clr.l    d2                   ;vertikal menuposition=0
    move.l   a1,a2                ;gem adresse til pointer
    tst.l    (a0)                 ;endnu en menu her?
    beq      setmenu1             ;nej: afslutning
    clr.l    (a1)+                ;forbered 'ikke flere menuer'
    move     d1,(a1)+             ;sæt x-position
    add.l    #70,d1               ;og forhøj
    move.l   #50,(a1)+            ;y-position og bredde
    move.l   #$a0001,(a1)+        ;højde og flag
    move.l   (a0)+,(a1)+          ;menutitel
    lea      12(a1),a3
    move.l   a3,(a1)+             ;pointer på menupunkter
    clr.l    (a1)+                ;reserverede ord
    clr.l    (a1)+

itemloop:
    tst.l    (a0)                 ;sidste indførelse?
    beq      menuend              ;ja: menu færdig?
    lea      54(a1),a3
    move.l   a3(a1)+              ;pointer på næste punkt
    move.l   d2;(a1)+             ;x/y-position
    add      #10,d2               ;y-position +10
    move.l   #$5a000a,(a1)+       ;bredde/højde
    move     #$52,(a1)+           ;flag: normalt
    clr.l    (a1)+                ;ingen tilslutning
    lea      16(a1),a3
    move.l   a3,(a1)+             ;tekststruktur-pointer
    clr.l    (a1)+                ;ingen udfyldningsstruktur
    clr.l    (a1)+                ;ingen command, ingen undermenu
    clr.l    (a1)+                ;og ingen fortsættelse
```

```
    move    #$1,(a1)+           ;sæt tekststruktur: farver
    clr     (a1)+               ;modus 0
    move.l  #$50003,(a1)+       ;x/y-position
    clr.l   (a1)+               ;standard-tegnsæt
    move.l  (a0)+,(a1)+         ;tekstpointer
    clr.l   (a1)+               ;ingen fortsættelse

    bra     itemloop            ;næste punkt...

menuend:                        ;eventuel overgang til næste menu
    clr.l   -54(a1)             ;slet pointer på næste punkt
    tst.l   (a0)+               ;forhøj tabelpointer
    tst.l   (a0)                ;endnu en menu her?
    beq     setmenu1            ;nej: færdig
    move.l  a1,(a2)             ;pointer på næste menu
    bra     menuloop            ;og fortsæt

setmenu1:                       ;* initialiser menu (som før)
    move.l  intbase,a6          ;Ituitions-basisadresse til A6
    move.l  windowhd,a0         ;vinduesstruktur i A0
    lea     menu,a1             ;pointer på menustruktur
    jsr     SetMenuStrip(a6)
    rts
```

Til programmet hører endnu tre ting: den lagerplads, der skal anvendes til strukturen, tabellen over tekstpointere og teksten selv. Her et eksempel:

```
mentab:
    dc.l  menu1                 ;første menutitel
    dc.l  mp11,mp12,mp13        ;menupunkter
    dc.l  0                     ;afslutning af menu 1

    dc.l  menu2                 ;anden menutitel
    dc.l  mp21,mp22,mp23        ;menupunkter
    dc.l  0                     ;afslutning på menu 2

    dc.l  0                     ;afslutning på hele menuen

;** menu-tekster **

menu1: dc.b "Menu 1",0
mp11:  dc.b "Punkt11",0
mp12:  dc.b "Punkt12",0
mp13:  dc.b "Punkt13",0
```

```
menu2: dc.b  "Menu2",0
mp21:  dc.b  "Punkt 21",0
mp22:  dc.b  "Punkt 22",0
mp23:  dc.b  "Punkt 23",0
even

;** plads til menustruktur **

menu:  blk.w 500
```

*Husk, at den reserverede lagerplads er stor nok til menustrukturen og forøg eventuelt angivelsen "blk.w 500"i den beregnede værdi.*

Hvis man anvender programmet og vil indbygge nogle specialiteter i menuen (f.eks. tastaturkommandoer), så kan man foretage ekstra indførelser i den opståede menustruktur-tabel, efter programmet er kørt. Man finder det tilsvarende ord (resp byte eller langord) i tabellen, hvis man går således frem:

For at finde tastaturkommando-byte'n for 2. indførelse i 1. menu skal man regne således:

**Adresse = begyndelsesadresse + menu*30 + (indførelse-1)*54 + 26**

det vil i eksemplet sige:

**adresse = menu + 30 + 54 + 26**
**= menu + 110**

26 er afstanden fra begyndelsen af MenuItem-strukturen til den søgte byte, kommandobyte'n. Gennem en række MOVE-kommandoer kan man tilpasse den færdige menu. I øvrigt: i ovenstående eksempel skal den tilsvarende flagbit være sat, for at tastatur-kommandoen kan 0.00

Nu igen til vinduet. Det er ganske rart at have et vindue, man kan forandre eller lukke. At kunne udlæse tekst i vinduet, er også ønskværdigt.

## 7.6. Tekstudlæsning

Intutitons tekstudlæsningsfunktion er meget men med vore allerede erhvervede kundskaber. Kald funktionen PrintIText med offset -216 og overgive den nogle parametre, som er:

**I A0** En pointer på vinduets RastPort, som findes i vinduesstrukturen.

**I A1** En pointer strukturen af den tekst, der skal udlæses.

**I D0** x-positionen.

**I D1** y-positionen af teksten i vinduet.

Det er let at foretage angivelserne af x- og y-position. Også strukturen, som vi skal opbygge til teksten, er ikke noget mysterium, da vi allerede har anvendt den to gange (requestere og menuer).

*Et nyt begreb er derimod vinduets RastPort, som er en tillægsstruktur, der beskriver det tilsvarende vindue. Adressen kræves af nogle af Intuitions funktioner.*

Pointeren på RastPort'en ligger i vinduesstrukturen fra den 50. byte. Vi skriver:

```
        move.l  windowhd,a0       ;adresse på vinduesstrukturen
        move.l  50(a0),a0         ;RastPort-adresse til A0
```

og har således fremskaffet adressen på RastPort'en. Til at udlæse en tekst behøver vi blot en rutine, der ser sådan ud, når vi overgiver x- og y-positionen i D0 og D1 samt tekststrukturens adresse i A1:

```
PrintIText = -216

        ...
print:
        move.l  intbase,a6        ;Intuitions-basisadresse til A6
        move.l  windowhd,a0       ;adresse på vinduesstrukturen
        move.l  50(a0),a0         ;RastPort-adresse til A0
        jsr     PrintIText(a6)    ;kald funktion
        rts
```

Man kan gennemprøve rutinen, ved at udlæse requesterens tekst, for hvilken der stadig er en struktur i programmet. For at opnå det, skriver man følgende linier foran label'en "loop":

```
        lea     btext,a1          ;pointer på tekststruktur i A1
        move.l  #10,d0            ;x-position
        move.l  #30,d1            ;y-position af teksten
        bsr     print             ;udlæs tekst
```

Og straks står teksten i vinduet. Hvis den ikke fremkommer, så undersøg den farve, der er angivet for teksten i tekststrukturen, og kontroller, om den er 0. Hvis ikke ændres den til 3, og teksten vil ved næste programstart fremkomme med rødt.

## 7.7. Images

"Image" er betegnelsen for en tegning i et rektangulært felt, som er defineret bitvis. Eksempler på sådanne images er diskettesymbolerne på Intutitions-screen'en eller system-gadgets på screen- og vindueskanterne, som kan udløse funktioner, når man klikker sig ind på dem.

Det kvadrat, hvori tegningen ligger, kan have vilkårlig størrelse. Men da der kræves en bit, for hver pixel indenfor kvadratet, er det ikke tilrådeligt at programmere images så store som skærmbilledet. Nøjes med ca. 32x16 bits, hvad der tillader images på omtrent 3x1 cm.

Sådanne images kan vises på forskellig måde, som man har set for gadgets i vinduerne. Der findes en Intuitions-funktion, der klarer tegningen af et image.

Det er DrawImage-funktionen med offset -114, som kræver 4 parametre:

**I A0** adressen på den RastPort, hvor billedet skal tegnes. Hvordan man får fat i adressen, har vi set under tekst-funktionen.

**I A1** kommer adressen på strukturen af det billede, der skal vises.

**I D0** kommer den relative x-position.

**I D1** kommer den relative y-position af tegningen.

Hvis vi forudsætter, at vi kun har et vindue, kan vi skrive en simpel underrutine, som vi overgiver adressen på image-strukturen i A1, og i D0, og D1 den position, hvortil tegningen skal bringes:

```
DrawImage  = -114

    ...
draw:                              ;* tegn image
    move.l  intbase,a6             ;Intuitions-basisadresse til A6
    move.l  windowhd,a0            ;pointer på vinduesstruktur
    move.l  50(a0),a0              ;så RastPort-adresse til A0
    jsr     DrawImage(a6)          ;tegn image
    rts
```

Så behøver vi den omtalte struktur af image't, som består af 9 indførelser med følgende betydning:

De første to indførelser er to ord, der angiver afstanden i x- og y-retning fra de angivne koordinater, hvori image't skal tegnes. Vi skriver blot to nuller:

```
image:
        dc.w    0,0             ;x- og y afstand
```

Derefter følger to ord, der angiver image'ts bredde og højde i pixels. Hvis vi vil tegne et image bestående af 32x13 punkter, så angiver vi:

```
        dc.w    32,13           ;bredde og højde af image
```

Det næste ord angiver, hvor mange plane'r tegningen skal have. Ønskes et billede, der kun skal vises i to farver, angives et 1; ved flere farver et tilsvarende større tal. Skal der arbejdes med flere farver, skal der angives tilsvarende flere data til image'ts mønster i det egentlige bitmønster. Lad os derfor blive ved 1:

```
        dc.w    1               ;et bitplane: 2↑1=2 farver
```

Så følger et langt ord, der peger på image'ts egentlige data:

```
        dc.l    imgdata         ;pointer på image-data
```

Videre går det med to bytes, hvis betydning er meget interessante. Den første byte hedder PlanePick, og angiver, i hvilke planes af vinduet eller screen'en der skal tegnes, dvs. image'ts data skal fremkomme. Da der kun skal vises et plane, spørges altså om vinduets bitplane. Informationen ligger i den pågældende byte's bits, idet bit 0 står for plane 0, bit 1 for plane 1 osv. Med angivelsen defineres den farve, hvori billedet skal vises. Hvis vi angiver et 2, bliver der sat en rød pixel for hver sat bit i image.

```
        dc.b    1               ;tegning rød: plane 1
```

Den anden byte med navnet PlaneOnOff er et interessant supplement. Her svarer hver bit af vindues-bit-plane't til samme nummer. Af interesse er kun de bits, hvor de tilsvarende er slettet i PlanePick-byte'n. Hvis bit'en er sat i PlaneOnOff, sættes hver bit af image't i det tilsvarende plane, ellers slettes den. For at ingen sat bit af image't vises i hvidt, angiver vi her et 1: alle bits i image't, der ikke er sat, anbringes i plane 1 og således i hvidt.

```
        dc.b    1               ;baggrund hvid
```

Og straks står teksten i vinduet. Hvis den ikke fremkommer, så undersøg den farve, der er angivet for teksten i tekststrukturen, og kontroller, om den er 0. Hvis ikke ændres den til 3, og teksten vil ved næste programstart fremkomme med rødt.

## 7.7. Images

"Image" er betegnelsen for en tegning i et rektangulært felt, som er defineret bitvis. Eksempler på sådanne images er diskettesymbolerne på Intutitions-screen'en eller system-gadgets på screen- og vindueskanterne, som kan udløse funktioner, når man klikker sig ind på dem.

Det kvadrat, hvori tegningen ligger, kan have vilkårlig størrelse. Men da der kræves en bit, for hver pixel indenfor kvadratet, er det ikke tilrådeligt at programmere images så store som skærmbilledet. Nøjes med ca. 32x16 bits, hvad der tillader images på omtrent 3x1 cm.

Sådanne images kan vises på forskellig måde, som man har set for gadgets i vinduerne. Der findes en Intuitions-funktion, der klarer tegningen af et image.

Det er DrawImage-funktionen med offset -114, som kræver 4 parametre:

**I A0** adressen på den RastPort, hvor billedet skal tegnes. Hvordan man får fat i adressen, har vi set under tekst-funktionen.

**I A1** kommer adressen på strukturen af det billede, der skal vises.

**I D0** kommer den relative x-position.

**I D1** kommer den relative y-position af tegningen.

Hvis vi forudsætter, at vi kun har et vindue, kan vi skrive en simpel underrutine, som vi overgiver adressen på image-strukturen i A1, og i D0, og D1 den position, hvortil tegningen skal bringes:

```
DrawImage  = -114

    ...
draw:                               ;* tegn image
    move.l  intbase,a6              ;Intuitions-basisadresse til A6
    move.l  windowhd,a0             ;pointer på vinduesstruktur
    move.l  50(a0),a0               ;så RastPort-adresse til A0
    jsr     DrawImage(a6)           ;tegn image
    rts
```

Så behøver vi den omtalte struktur af image't, som består af 9 indførelser med følgende betydning:

De første to indførelser er to ord, der angiver afstanden i x- og y-retning fra de angivne koordinater, hvori image't skal tegnes. Vi skriver blot to nuller:

```
image:
    dc.w    0,0         ;x- og y afstand
```

Derefter følger to ord, der angiver image'ts bredde og højde i pixels. Hvis vi vil tegne et image bestående af 32x13 punkter, så angiver vi:

```
    dc.w    32,13       ;bredde og højde af image
```

Det næste ord angiver, hvor mange plane'r tegningen skal have. Ønskes et billede, der kun skal vises i to farver, angives et 1; ved flere farver et tilsvarende større tal. Skal der arbejdes med flere farver, skal der angives tilsvarende flere data til image'ts mønster i det egentlige bitmønster. Lad os derfor blive ved 1:

```
dc.w        1           ;et bitplane: 2↑1=2 farver
```

Så følger et langt ord, der peger på image'ts egentlige data:

```
dc.l        imgdata     ;pointer på image-data
```

Videre går det med to bytes, hvis betydning er meget interessante. Den første byte hedder PlanePick, og angiver, i hvilke planes af vinduet eller screen'en der skal tegnes, dvs. image'ts data skal fremkomme. Da der kun skal vises et plane, spørges altså om vinduets bitplane. Informationen ligger i den pågældende byte's bits, idet bit 0 står for plane 0, bit 1 for plane 1 osv. Med angivelsen defineres den farve, hvori billedet skal vises. Hvis vi angiver et 2, bliver der sat en rød pixel for hver sat bit i image.

```
dc.b        1           ;tegning rød: plane 1
```

Den anden byte med navnet PlaneOnOff er et interessant supplement. Her svarer hver bit af vindues-bit-plane't til samme nummer. Af interesse er kun de bits, hvor de tilsvarende er slettet i PlanePick-byte'n. Hvis bit'en er sat i PlaneOnOff, sættes hver bit af image't i det tilsvarende plane, ellers slettes den. For at ingen sat bit af image't vises i hvidt, angiver vi her et 1: alle bits i image't, der ikke er sat, anbringes i plane 1 og således i hvidt.

```
dc.b        1           ;baggrund hvid
```

Som sidste indførelse i strukturen kommer et langt ord, der peger på et eventuelt ekstra image. Vi har ingen brug for det og sætter derfor det lange ord lig nul:

```
dc.l        0                 ;ikke flere image'r
```

Her er image-strukturen:

```
image:
    dc.w    0,0               ;x- og y-afstand
    dc.w    32,13             ;bredde og højde af image
    dc.w    1                 ;et bitplane: 2↑1=2 farver
    dc.l    imgdata           ;pointer på image-data
    dc.b    1                 ;tegning rød: plane 1
    dc.b    1                 ;baggrund: hvid
    dc.l    0                 ;ikke flere image'r
```

Så til selve image-dataene. Til hver række punkter angives med et langt ord eller flere et mønster, hvor der for hvert punkt af billedet, der skal sættes, er sat den tilsvarende bit. Det gentages så tit, som angivet af billedets højde. Hver linies data skal begynde ved ordgrænserne, dvs. på lige adresser.

I eksemplet er det let at opbevare dataene, idet vi har angivet 32 punkter horisontalt, hvad svarer nøjagtigt til et langt ord. Til at programmere billedet er det naturligt at anvende den binære repræsentation af dataene.

Lad os som eksempel tage en afbryder i "OFF"-stilling. Formen er valgt med god grund, så man bør skrive den ind. Vi skal nemlig i et senere kapitel om gadgets se, hvorledes vi kan omstille denne afbryder. Her er dataeksemplerne for afbryder-image't:

```
imgdata:                      ;data for afbryder i OFF-stilling
    dc.l %00000000000000000000000000000000
    dc.l %00000000000000000000111000000000
    dc.l %00011101110111000011111000000000
    dc.l %00010101000100000011111000000000
    dc.l %00010101100110000001111000000000
    dc.l %00011101000100000011100000000000
    dc.l %00000000000000000111000000000000
    dc.l %00000000000000000111000000000000
    dc.l %00000000000111111111100000000000
    dc.l %00000000001111111111110000000000
    dc.l %00000000001111111111110000000000
    dc.l %00000000000110000001100000000000
    dc.l %00000000000000000000000000000000
```

Er disse data indtastet rigtigt, kan man afprøve dem, ved at lade dem vise. man skriver før "loop"-labelen:

```
move.l  image,a1        ;pointer på image-struktur
move    #30,d0          ;x-position i vinduet
move    #50,d1          ;y-position
bsr     draw            ;tegn image
```

Forhåbentlig er resultatet tilfredsstillende? Afbryderen møder vi igen, når der skal omstilles til "ON" under gadgets. Men først skal vi beskæftige os, med en anden metode til at tegne i vinduerne.

## 7.8. Borders

Border betyder på dansk rammer, hvilket er et lidt upræcist udtryk for denne funktion. Med den kan man nemlig tegne vilkårligt mange sammenhængende linier, ligegyldigt af hvilken længde eller vinkel.

Funktionen indsættes af Intuition, f.eks. for at tegne rammerne om vinduer og screens. Den bruges også til at frembringe rammer om mønstre eller tekster, især ved de såkaldte streng-gadgets. Herom senere.

Betjeningen af funktionen er ret simpel. Vi bruger først Intuitions-funktionen DrawBorder med offset -108, som vi overgiver med 4 parametre:

**I A0** RastPort-adressen på det udlæsningsmedium, hvori linierne skal tegnes. Hertil anvendes igen vort vindue.

**I A1** overgives adressen på border-strukturen, som vi skal se nærmere på.

**I D0** skal de relative x-koordinater, hvorfra positionsværdierne i x/y-koordinatlisten for linierne beregnes

**I D1** de relative y-koordinater. Relativ betyder også, at angivleserne er i forhold til vinduets øvre venstre hjørne

Skriv en lille underrutine, som ved kald skal overgive strukturadressen i A1 og x/y-koordinaterne i D0 og D1. Principielt tegnes i det vindue, hvis strukturadresse står i "windowhd".

```
DrawBorder = -108

    ...
borderdraw:                     ;* tegn flerdobbelt linie
    move.l  intbase,a6          ;Intuitions-basisadresse til A6
```

```
        move.l  windowhd,a0      ;pointer på vinduesstruktur
        move.l  50(a0),a0        ;så RastPort-adresse til A0
        jsr     DrawBorder(a6)   ;tegn linier
        rts
```

Nu til den struktur, border'en kræver. Der anføres 8 parametre i listen, nemlig følgende:

Først angives i to ord den vertikale og den horisontale afstand fra de koordinater, der er angivet i funktionskaldet. For ikke at tabe overblikket over de forskellige afstandsangivelser, skrives et nul hvert sted:

```
border:
        dc.w    0                ;horisontal afstand
        dc.w    0                ;vertikal afstand
```

Derefter følger igen to bytes, hvori farverne bestemmes. Vi vælger en rød ramme:

```
        dc.b    3                ;rød ramme
        dc.b    0                ;baggrund (ubenyttet)
```

Som det ses, er det ikke vigtigt at angive baggrundsfarve. Alligevel kan vælges mellem to modi, hvori linierne kan tegnes. Den følgende byte bestemmer modus. Er den 0, tegnes hver linie helt i den angivne farve, uafhængigt af, hvad der var før. Denne modus kalder man JAM1. Den anden mulighed, hvor begge farveangivelser ignoreres, er XOR-modus. Her bliver alle punkter, der ligger nedenunder linien, inverteret i farveværdi. Således bliver et hvidt til et sort eller et blåt til et rødt punkt. Denne modus indstilles med et 2. I eksemplet vil vi hellere anvende JAM1-modus:

```
        dc.b    0                ;modus: JAM1 (2=XOR)
```

Nu angives i endnu et ord, hvor mange koordinatpar, som skal foreligge i listen. Da ordet igen skal ligge på en lige adresse, bliver først indtastet en "even"-kommando, og derefter antallet af par. Læg mærke til, at tre punkter er nødvendige for at tegne to linier: begyndelses-, hjørne- og slutpunkt. For at tegne en firkantet ramme, behøver vi altså fem par:

```
        dc.b    5                ;5 xy-par skal forbindes
```

Dernæst følger pointeren på koordinattabellen, hvori de punkter, der skal forbindes, er defineret:

```
        dc.l    koord            ;pointer på koordinattabel
```

Afslutningen af border-strukturen dannes af et langt ord, der kan pege på en ny border-struktur. Hvis der ikke er nogen følgende struktur, tilføjes et nul. Pointeren er helt igennem nyttig, da man f.eks. kan tegne en tofarvet rand om et eller andet, ved at forbinde to selvstændige border-struturer, og på den måde fremhæve det. Vi vil dog ikke anvende pointeren til vort eksempel:

```
        dc.l    0               ;ingen yderligere struktur
```

Det var border-strukturen. Så mangler koordinatlisten. Her er et eksempel med en liste, der består af fem par, som tegner et rektangel. Jeg vil anbefale, at man anvender disse værdier, da vi skal bruge dem i et senere programeksempel.

```
koord:                          ;koordinater til rektangulær ramme
        dc.w    -2,-2
        dc.w    80,-2
        dc.w    80,9
        dc.w    -2,9
        dc.w    -2,-2
```

Her er de samlede border-strukturer:

```
border:
        dc.w    0               ;horisontal afstand
        dc.w    0               ;vertikal afstand
        dc.b    3               ;rød ramme
        dc.b    0               ;baggrund (ubenyttet)
        dc.b    0               ;modus: JAM1 (2=XOR)
        dc.b    5               ;5 xy-par skal forbindes
        dc.l    koord           ;pointer på koordinattabel
        dc.l    0               ;ingen yderligere struktur

koord:                          ;koordinater til rektangulær ramme
        dc.w    -2,-2
        dc.w    80,-2
        dc.w    80,9
        dc.w    -2,9
        dc.w    -2,-2
```

Når alt er indtastet, kan man prøve det igennem. Dertil skrives foran programmets "loop"-label:

```
        lea     border,a1       ;adresse på border-strukturen
        move    #20,d0          ;x-grundposition
```

```
move    #80,d1          ;y-grundposition
bsr     borderdraw      ;tegn ramme
```

Som det ses, kan Eiffeltårnet tegnes med denne funktion, hvis man har et passende antal koordinater. Men tilbage til de simple tegninger. Vi vil nu genoplive vore tegninger, dvs. lade dem blive styrbare med musen.

## 7.9. Gadgets

Begrebet gadgets er omtalt under opbygningen af vinduer. Her var det system-gadgets som lukkesymbolet, som man kan aktivere ved at klikke ind på dem, og som så eventuelt udløser en funktion i programmet.

*Foruden fastlagte gadgets, kan man selv designe og vise dem. Her byder Intuition på en mængde muligheder, som kan fremtrylle interessante effekter!*

Man skelner mellem fire typer gadgets:

- **Boolean-gadgets tjener til at vælge mellem ja og nej. Man klikker dem og aktiverer (ja), eller inaktiverer dem (nej).**
- **Streng-gadgets, der kan anvendes til indlæsning af tekster af forud fastsat længde.**
- **Integer-gadgets er en specialform af streng-gadgets og er til indlæsning af et decimaltal, hvis værdi Intuiton selv omregner til et langt ord, og giver tilbage til programmet.**
- **Proportional gadgets tillader valget af en analog værdi med musen, idet man skubber noget frem og tilbage.**

### 7.9.1. Boolean-gadgets

Lad os begynde med den simpleste type, boolean-gadgets. Sådan en gadget er f.eks. vinduets lukkesymbol, der kun kan skelne mellem "klikket ind" eller "ikke klikket ind". Vi vil trin for trin udvikle en gadget af denne type. Flagene osv. er stort set de samme for andre typer gadgets.

Hvert gadget behøver en struktur med 15 indførelser. Pointeren på strukturen bliver indført i strukturen til det vindue, screen eller requester, hvori gadget'en skal fremkomme. Til det formål var der reserveret et langt ord hvert sted, hvori vi hidtil har skrevet et 0. Hvis gadget strukturens adresse indføres her, så bliver de(t) pågældende gadget(s) vist, samtidig med at vinduet åbnes.

Indførelserne i gadget-strukturen skal ske i følgende rækkefølge:

Det første lange ord er en pointer, som i givet fald peger på den næste gadget, der skal installeres. Gadgets'ene ligger efter hinanden som perler på en snor, hvorunder den pointer, der skal indføres i window-strukturen, skal pege på den første gadget i kæden. Først vil vi kun bringe en gadget ind i vinduet, så vi skriver et nul:

```
gadget 1:
    dc.l   0          ;ikke flere gadgets
```

Så følger igen to ord, der bestemmer gadget'ens position i vinduet. Her er flere muligheder for positionsbestemmelse, som vi fastsætter i de flag, der følger efter. Men lad os gå ud fra en fast placeret gadget:

```
    dc.w   40         ;x-position og
    dc.w   50         ;y-position af gadget
```

Derefter kommer to ord, der bestemmer målene på gadget'ens "hit-box". Boxen angiver ikke gadget'ens synlige størrelse (den afhænger af de eventuelt angivne image-data), men størrelsen af det rektangel, der overvåges af Intuition. Hvis musens pointer bringes ind i boxen, og der trykkes på venstre knap, så aktiveres gadget'en. Hvis man klikker ind på dele af gadget'en, der ligger udenfor denne box, bliver det ignoreret.

```
    dc.w   32         ;bredde og
    dc.w   13         ;højde af hit-box
```

Nu følger det ord, hvori bits bestemmer gadget'ens særlige egnskaber. I bits 0 og 1 bestemmes, hvad der skal ske med gadget'en, når der klikkes på den (resp. dens hit-box). Betydningen af værdien i disse bits er følgende:

| **Bit 0,** | **1** | **Værdi** | **Navn** | **Betydning** |
|---|---|---|---|---|
| 0 | 0 | 0 | GADGHCOMP | **gadget'en inverteres** |
| 0 | 1 | 1 | GADGHBOX | **gadget'en omrammes** |
| 1 | 0 | 2 | GADGHIMAGE | **der fremkommer et andet image** |
| 1 | 1 | 3 | GADGHNONE | **ingen reaktion** |

Bit 2 bestemmer, om gadget'en skal bestå af en tegning eller en ramme: hvis den er sat (værdi +4), er det et billede, ellers en border.

Den næste bit bestemmer, om positionen af gadget'en skal være i forhold til den øverste, eller den underste kant af vinduet. Hvis den er sat (værdi +8), vurderes positionen relativt i forhold til den underste, ellers i forhold til den øverste kant. Den næste bit har samme betydning for den horisontale position: hvis den er sat (værdi +$10), er det en relativ, ellers en absolut placering.

*Bemærk, at ved definitionen af en relativ gadget, skal positionsangivelserne i de første ord af strukturen være negative, da den ønskede position ikke ligger nedenunder, men ovenover referencepositionen.*

Således kan man både bestemme en relativ og en absolut position af gadget'en. Et eksempel på en absolut placeret gadget er system-gadget'en til lukning af vinduet, en relativ gadget er symbolet for størrelsesændring.

Bredden og højden af gadget'en resp. hit-boxen kan bestemmes relativt i forhold til vinduesstørrelsen. Det angives med bit 5 for bredden (værdi +$20) og bit 6 for højden (værdi +$40). Også her gælder det, at en sat bit betyder relativ størrelse.

Bit 7 (værdi +$80) bevirker, når den er sat, at gadget'en bliver aktiv, når vinduet åbnes.

Bit 8 (værdi +$100) bestemmer, om gadget'en er valgbar. Hvis denne bit er sat, kan gadget'en ikke aktiveres.

Vi vælger den absolutte placering og størrelse, den inverterede afbildning af den aktiverede gadget og visning af gadget'en som image. Det betyder, at vi skal indføre værdien 4:

```
        dc.w    $4          ;flag: image, inverteres
```

Så følger et ord, hvis bits alle er flag. Disse flag kaldes Activation Flags, og bestemmer visse funktioner af gadget'en. Bits'enes betydning er følgende:

| Bit | Værdi | Navn | Betydning: meddelelse ved |
|---|---|---|---|
| 0 | 1 | RELVERIFY | bevirker, at gadget'en kun aktiveres, hvis den venstre museknap slippes over gadget'en. |
| 1 | 2 | GADGIMMEDIATE | lader gadget'en blive aktiveret med det samme, når den klikkes. |

| Bit | Værdi | Navn | Betydning: meddelelse ved |
|---|---|---|---|
| 2 | 4 | ENDGADGET | lader, hvis gadget'en ligger i en requester, denne afslutte ved valg og således forsvinde. |
| 3 | 8 | FOLLOWMOUSE | lader ved valgt gadget, til stadighed musens position melde, til den standses. Hermed laves en forskydning af en gadget med musen, hvis man ændrer gadget-positionen tilsvarende. |
| 4 | $10 | RIGHTBORDER | bevirker ved anvendelsen af borders, at den |
| 5 | $20 | LEFTBORDER | tilsvarende side altid bliver tilpasset |
| 6 | $40 | TOPBORDER | størrelsen af gadget'en, så den passer til |
| 7 | $80 | BOTTOMBORDER | rammen. |
| 8 | $100 | TOGGLESELECT | lader gadget'en ændre sin tilstand, hver gang der klikkes på den: aktiveret bliver til inaktiveret og omvendt. |
| 9 | $200 | STRINGCENTER | bestemmer ved streng-gadgets, hvordan strengen |
| 10 | $400 | STRINGRIGHT | skal vises, på midten eller med lige højrekant. Hvis ingen af disse bits er sat, vises strengen med lige venstrekant. |
| 11 | $800 | LONGINT | forvandler en streng-gadget til en integer-gadget (forklaring følger). |
| 12 | $1000 | ALTKEYMAP | bevirker, at en anden tastaturbelægning gælder ved strengindlæsning. |

Det var aktiveringsflagene. Vi vælger nu flagene TOGGLESELECT og GADGETIMMEDIATE til vort eksempel:

```
dc.w    $102        ;Activation
```

Det næste ord i gadget-strukturen bestemmer gadget-typen. Her er de enkelte bits betydning:

| Bit | Værdi | Navn | Betydning: meddelelse ved |
|---|---|---|---|
| 0 | 1 | BOOLGADGET | dette er en boolean-gadget |
| 1 | 2 | GADGETOO2 | ? |
| 2 | 4 | STRGADGET | streng- eller integer-gadget |
| 0+1 | 3 | PROPGADGET | proportional-gadget |
| System gadgets: | | | |
| 4 | $10 | SIZING | størrelsesændrings-gadget |
| 5 | $20 | WDRAGGING | forskydnings-gadget til vindue |
| 4+5 | $30 | SDRAGGING | det samme til screens |
| 6 | $40 | WUPFRONT | gadget til at "hente vindue frem" |
| 6+4 | $50 | SUPFRONT | gadget til at "hente screen frem" |
| 6+5 | $60 | WDOWNBACK | "sæt vindue bagest" |
| 6+5+4 | $70 | SDOWNBACK | "sæt screen bagest" |
| 7 | $80 | CLOSE | vindueslauknings-gadget |
| Typedefinitioner | | | |
| 12 | $1000 | REQGADGET | requester-gadget |
| 13 | $2000 | GZZGADGET | rand-gadget i GIMMEZEROZERO vindue |
| 14 | $4000 | SCRGADGET | screen-gadget, når sat |
| 15 | $8000 | SYSGADGET | system-gadget, når sat |

Vi vil have en boolean-gadget til eksempel og vælger dette med

```
dc.w    1           ;gadget-type: boolean
```

Så følger nogle pointere i gadget-strukturen. Den første pointer indeholder adressen på image- eller border-strukturen, som anvendes til visningen af gadget'en. Er det ikke nødvendigt med visning, skrives et nul. Vi vælger til vor gadget visningen som image, og indsætter derfor pointeren på image-strukturen, som vi har oprettet i kapitlet om images:

```
        dc.l    image           ;gadget-image
```

Den næste pointer bliver kun anvendt, når GADGHIMAGE-flaget er sat i strukturens flag-ord. Så bliver pointeren rettet mod endnu en struktur, der skal vises ved aktiveret gadget. Men når der anvendes en border-struktur til fremstilling af gadgets, må der også vises en border-struktur. Vi udelader visningen af et image mere, og indsætter derfor et nul:

```
        dc.l    0               ;ingen select-gadget
```

Den næste pointer peger på en struktur i den tekst, der skal udlæses med gadget'en. Hvis der ikke kræves nogen tekst, indføres et nul. Men vi vil have en tekst:

```
        dc.l    ggtext          ;gadget-tekst
```

Nu følger et langt ord, der bestemmer, hvilke gadgets, der skal inaktiveres, når det aktuelle aktiveres. Funktionen er endnu ikke i orden, så vi kan skrive et nul her:

```
        dc.l    0               ;intet exclude
```

Også næste pointer kan vi sætte på nul, da den kun anvendes ved streng- og proportional-gadgets. Med en af disse skal den pege på en speciel struktur til beskrivelse af denne gadgets egenskaber, som kaldes SpecialInfo.

```
        dc.l    0               ;ingen Special Info
```

Så følger et ord, hvori vi skriver det løbende nummer på gadget'en, der anvendes som gadget-identifikation (ID).

```
        dc.w    1               gadget-ID
```

Sidst endnu et langt ord, der ikke har nogen funktion og derfor sættes til nul

```
        dc.l    0               ;UserData (ignoreres)
```

Det var det. Her er hele Gadget-strukturen:

```
gadget1:
        dc.l    0               ;ikke flere gadgets
        dc.w    40              ;x-position og
        dc.w    50              ;y-position af gadget
        dc.w    32              ;bredde og
```

```
        dc.w    13          ;højde af hit-box
        dc.w     4          ;flag: image, inverteres
        dc.w  $102          ;activation-flag
        dc.w     1          ;gadget-type: boolean
        dc.l image          ;gadget-image
        dc.l     0          ;intet selct-gadget
        dc.l ggtext         ;gadget-tekst
        dc.l     0          ;intet exclude
        dc.l     0          ;ingen Special Info
        dc.w     1          ;gadget-ID
        dc.l     0          ;Userdata (ignoreres)
```

Vi har allerede en struktur for image't, som vi kan anvende. Nu skal vi definere den tekst, der skal stå under vor gadget.

Den gadget ser ud som en afbryder, derfor foreslår jeg påskriften "switch". Strukturen af teksten vil være følgende:

```
ggtext:
        dc.b 1,0            ;farver
        dc.b 1              ;modus
even
        dc.w -8,14          ;x/y-position
        dc.l 0              ;standard typesæt
        dc.l swtext         ;pointer på tekst
        dc.l 0              ;ingen tekst mere

swtext:
        dc.b "Switch",0
even
```

Når man har skrevet alt dette ind, saver man det for en sikkerheds skyld, assembler det, og starter det. Man kan klikke på afbryderen, hvad der får den til at blive vist inverteret. Hvis man klikker på den en gang til, bringes den atter tilbage til det normale.

Man kan lave nogle eksperimenter med strukturen ved hjælp af tabellerne. Ved at ændre flaget fra 4 til 5 opnås, at gadget'en bliver indrammet, når den aktiveres. Eller man kan prøve med sat RELVERIFY-bit (bit 0: +1) i Activation-Flags ordet. Så kan man stille musens pointer på gadget'en og trykke på tasten: den bliver aktiveret. Hvis man holder musens knap nedtrykket, og bevæger musen, vil man se, at aktiveringen atter ophæves, når man forlader hit-boxen. Hvis man ved en fejl har klikket ind på gadget'en, behøves kun at køre ud af feltet, for at forhindre aktiveringen.

Prøv at omstille afbryderen. Vi behøver blot en imagestruktur, for den afbrudte tilstand. Så sætter vi i gadget-strukturen pointeren på denne struktur og bagefter på den normale image-struktur. Desuden ændrer vi flag-ordet til 6, hvad der bevirker visningen af det andet billede ved aktivering.

Her er image-strukturen for den tilkoblede afbryder:

```
image2:
    dc.w 0,0                ;intet offset
    dc.w 32,13              ;32x13 pixels
    dc.w 1                  ;modus 1
    dc.l  imgdata2          ;pointer på dataene
    dc.b  2,1               ;samme farver som før
    dc.l 0                  ;ellers intet

imgdata2                    ;data for afbryder i ON-stilling
    dc.l  %00000000000000000000000000000000
    dc.l  %00000000011100000000000000000000
    dc.l  %00000000111110000011101001000000
    dc.l  %00000000111110000010101101000000
    dc.l  %00000000011110000010101011000000
    dc.l  %00000000000111000011101001000000
    dc.l  %00000000000011100000000000000000
    dc.l  %00000000000001110000000000000000
    dc.l  %00000000000111111111000000000000
    dc.l  %00000000001111111111110000000000
    dc.l  %00000000001111111111110000000000
    dc.l  %00000000000110000001100000000000
    dc.l  %00000000000000000000000000000000
```

Sådan vises gadget'ens tilstand. "Afbryderstillingen" viser gadget'ens tilstand: hvis den er aktiveret, står afbryderen på ON, ellers på OFF.

Så meget om boolean gadgets. Hvad der eventuelt ikke er forklaret, kan man sandynligvis finde ud af, ved at eksperimentere. Vi går videre til streng-gadgets, som man kan lave pæne ting med.

## 7.9.2. Streng-gadgets

Man bedes forestille sig, at man vil indlæse data fra diskette i et program. For at opfordre brugeren til at indtaste filnavnet, må der udlæses en passende tekst, der indbyder til at indlæse navnet, og dernæst skal der kaldes en indlæsningsrutine, der vurderer tastaturindlæsningen.

Simplere og elegantere ville det være, at anvende en streng-gadget. Denne funktion tillader en behagelig indlæsning og/eller ændring af en kort tekst med valgfri indramning. Det er også muligt med en "undo"-funktion, hvor det gamle indhold af gadget'en, altså den gamle tekst, genfremkaldes ved at trykke på højre Amiga-tast og "Q".

*Endnu en væsentlig detalje er, at størrelsen af teksten og tillige af indlæsningsfeltet lader sig variere. Hvis teksten er længere end indlæsningsfeltet er bredt, så bliver teksten skubbet frem og tilbage af det lille vindue, når man er nået til kanten med cursortasterne, eller under normal indlæsning.*

Alt det er argumenter for streng-gadgets, som retfærdiggør anvendelsen af gadgets. Som en ekstra godbid findes der også metoder til at begrænse indlæsningen til cifre, for at muliggøre indlæsningen af et tal. Forvandlingen af ciffertegnene til et binært tal i et langt ord, overtages af Intuition, hvad der er en stor lettelse, netop for maskinsprogs-programmører. En sådan specialiseret streng-gadget kalder man en integer-gadget.

Opbygningen af strukturen er, i udstrakt grad, den samme som af boolean gadgets. Der er kun to punkter, der er væsenligt forskellige:

I strukturens type-ord skal sættes et 4, for at indstille denne type på streng-gadget (STRGADGET).

Pointeren på SpecialInfo-strukturen er nødvendig. Her sætter vi en pointer, der peger på den StringInfo-struktur, der først skal oprettes.

Angivelse af bredde og højde på gadget'en i gadget-strukturen får nu også en anden betydning end før. Man angiver igen det område, hvorhen man skal bringe musen, og trykke på knappen for at aktivere streng-gadget'en. Alle disse angivelser bliver desuden bestemmende for den måde, teksten vises på. Værdien udgør målene på den såkaldte "container", hvori teksten bliver udlæst. Det anbefales at tegne en ramme omkring containeren med border-funktionen, for at man tydeligt kan se grænserne.

Hvis teksten er længere end containeren, vises kun en del af den, som man kan skubbe gennem containeren, der kan betragtes som et vindue, ved hjælp af tekstindlæsning eller tryk på venstre/højre cursortast. De tegn bliver altid skubbet ind på cursorpositionen. Dvs. resten af teksten bliver skubbet en plads fremad. For at ændre et tegn, må man først slette det. De mulige funktioner for tekstbehandling er følgende:

**Cursortaster til venstre/højre**
Bevæger cursoren over den aktuelle tekst, skubber i givet fald teksten gennem containeren.

**Cursortaster med Shift**
Sætter cursoren ved begyndelsen, resp. slutningen af teksten.

**DEL**
Sletter tegnet under cursoren

**BackSpace**
Sletter tegnet til venstre for cursoren.

**Return**
Afslutter tekstindlæsningen.

**Amiga til højre + Q**
Sætter teksten tilbage på det oprindelige indhold (undo).

StrengInfo-strukturen besidder kun få parametre:

Først angives en pointer, som peger på det lagerområde, der anvendes til opbevaring af den indtastede tekst. Lagerbufferen må vælges passende stor.

```
strinfo:
        dc.l    strpuffer     ;pointer på tekstbufferen
```

Så følger en pointer på en såkaldt undo-buffer. Pointeren og bufferen er kun påkrævet, når man ønsker undo-funktionen. Bufferen skal være mindst lige så stor som tekstbufferen. Hver gang man kalder streng-gadget funktionen, bliver indholdet af tekstbufferen kopieret over i denne buffer. Hvis man under indlæsningen trykker på højre Amiga-tast og "Q", genoprettes den gamle tilstand af teksten, ved at indholdet af undo-bufferen igen kopieres over i tekstbufferen. Hvis man anvender flere streng-gadgets i et program, kan man benytte samme undo-buffer til dem allesammen, da man kun kan benytte n streng-gadget ad gangen.

```
        dc.l    undo          ;pointer på undo-buffer
```

Følgende ord indeholder cursorpositionen i teksten. Man bør sætte ordet på nul, så man kan se hele teksten, resp. dens begyndelse, ved kald af det indtastede.

```
        dc.w    0             ;cursor-position
```

Som næste ord angives det maksimale antal tegn, der skal indtastes. Hvis man efter at have indlæst dette antal, skriver et til, så giver billedskærmen et kort blink for at melde, at det ikke går længere. Antallet af tegn og den reser-

verede plads til indlæsningsfeltet, skal ikke ubetinget stemmme overens, da teksten bliver skubbet videre under indtastningen.

```
        dc.w  10            ;max. char
```

I det følgende ord indføres, fra hvilket tegn af teksten i bufferen, teksten i containeren, skal udlæses. Her bør man anvende et nul, for at få begyndelsen på teksten at se.

```
        dc.w  0             ;udlæs tekst fra dette tegn
```

Så følger yderligere fem ord, som skrives i, af Intuition, og derfor ikke behøver at initialiseres. Det er tilstrækkeligt at sætte dem alle på nul. Ordnen får følgende informationer:

```
        dc.w  0             ;tegnposition i undo-buffer
        dc.w  0             ;antal af tegn i tekst-buffer
        dc.w  0             ;antal tegn synlige i container
        dc.w  0             ;horisontalt container-offset
        dc.w  0             ;vertikalt container-offset
```

Herefter følger to lange ord, der initialiseres og beskrives af Intutition:

```
        dc.l  0             ;pointer på den aktuelle RastPort
        dc.l  0             ;langord med indlæsningsværdien (ved
                            integergadgets)
```

Afslutningen udgøres af en pointer, som sættes på en tastaturtabel, når man har sat gadget'ens ALTKEYMAP.

```
    dc.l 0                  ;standard-tastaturbelægning
```

Her er den samlede StringInfo-struktur:

```
    strinfo:
        dc.l  strpuffer     ;pointer på tekstbuffer
        dc.l  undo          ;pointer på undo-buffer
        dc.w  0             ;cursor-position
        dc.w  10            ;max. char
        dc.w  0             ;udlæs tekst fra dette tegn
        dc.w  0             ;tegnposition i undo-buffer
        dc.w  0             ;antal tegn i tekst-buffer
        dc.w  0             ;antal tegn synlige i container
        dc.w  0             ;horisontalt container-offset
```

```
        dc.w    0               ;vertikalt container-offset
        dc.l    0               ;pointer på aktuelle RastPort
        dc.l    0               ;langord med indlæsningsværdien (ved
                                 integer-gadgets)
        dc.l    0               ;standard-tastaturbelægning
```

og så blot teksten og undo-bufferen:

```
strpuffer:
        dc.b    ”Hallo !”,0,0,0

undo:
        dc.l    0,0,0,0
even
```

Når man har indlæst disse linier, kan man enten ændre den gamle gadget-struktur tilsvarende, eller opbygge en ny. For stadig at have afbryderen med i billedet og kunne benytte den, anbefales at opbygge en ny gadget-struktur. Den første pointer i den gamle struktur skal ændres fra 0 til ”gadget1”, og den nye struktur indføjes. Her er et struktureksempel til streng-gadget'en, med de ovenfor angivne indførelser:

```
gadget1:                        ;* struktur til streng-gadget
        dc.l    0               ;ikke flere gadgets
        dc.w    20,80           ;position
        dc.w    80,10           ;bredde/højde af containeren
        dc.w    0               ;flag: normale
        dc.w    2               ;activation ($802 til longint)
        dc.w    4               ;type: streng-gadget
        dc.l    border          ;pointer på indramning
        dc.l    0               ;ingen select-tegning
        dc.l    0               ;ingen tekst
        dc.l    0               ;intet exclude
        dc.l    strinfo         ;pointer på StringInfo-struktur
        dc.w    2               ;gadget-ID
        dc.l    0               ;ingen User-data

border:                         ;* border til indramning af container
        dc.w    0,0             ;intet offset
        dc.b    3,3             ;farve: rød
        dc.b    0               ;modus: JAM1
        dc.b    5               ;5 xy-par
        dc.l    koord           ;pointer på koordinat-tabel
        dc.l    0               ;ingen fortsættelse
```

```
koord:                      ;* koordinater til indramning
       dc.w   -2,-2         ;begyndelse øverst til venstre
       dc.w   80,-2         ;øverst til højre
       dc.w   80,9          ;underst til højre
       dc.w   -2,9          ;underst til venstre
       dc.w   -2,-2         ;til begyndelsen
```

Med de angivne data, fremkommer i vinduet et rødt rektangel (borderen) hvori der står teksten "Hallo !". Teksten kan ændres ved at klikke ind på feltet og editere, efter at cursoren er kommet til syne. Har man skrevet noget forkert, kan man få "Hallo !" til at komme frem igen med undo-funktionen (Amiga til højre og "Q").

Når man har indtastet noget, og derefter inaktiveret gadget'en ved at trykke return, eller klikke uden for feltet (cursoren forsvinder), kan man prøve at afbryde programmet. Så kan man indtaste "q strinfo" i SEKA , og betragte parametrene samt teksten og undo-bufferen.

Vi ændrer activation-flaget til $802 og "strpuffer" til "dc.l 0,0,0,0", assembler og starter programmet igen. Igen kan man indtaste noget, efter at have aktiveret streng-gadget'en, men kun cifre. Et forsøg på at indtaste et bogstav, bliver afvist med blinkende billedskærm.

Man indtaster et tal, og afslutter programmet, efter at have inaktiveret gadget'en. Når man ser på StringInfo-strukturen ved hjælp af "q info", kan man genfinde den lige indtastede værdi med hexadecimal skrivemåde i det ottende langord ("q strinfo+32").

Vi kommer så til de gadgets, hvormed man kan indlæse, eller ændre en analog værdi ved forskydning af et symbol: proportional-gadgets.

## 7.9.3. Proportional-gadgets

Læseren kender sikkert fortrinnene ved en skyderegulator frem for en drejeregulator, f.eks. i HiFi-anlæg eller brødristere. Man kan se den indstillede værdi på dem, særlig hvis der ligger flere regulatorer ved siden af hinanden (f.eks. graphic equalizere i forstærkere). Sådanne skyderegulatorer kan vi vise på Amigaens billedskærm, og betjene med musen, hvad der giver meget smukke udformningsmuligheder af programmerne.

Det gennemføres med gadgets. Gadget-typen kan forskyde et symbol horisontalt og/eller vertikalt indenfor en ramme. Størrelsen af rammen og skyderen kan udformes variabelt. F.eks. kan rammestørrelsen være relativ i for-

hold til vinduesstørrelsen, og variere sammen med ændringer af vinduet. Skyderen kan indstilles så dens størrelse vokser, eller skrumper ind, inden for rammen samtidig med indstillingsbredden.

Alle muligheder (og det var ikke en gang allesammen) kan bedst forklares eller afprøves med et eksempel. Til det formål oprettes et proportional-gadget, der kan forskydes horisontalt.

Vi behøver igen en gadget-struktur, der har samme opbygning som de tidligere. For at tydeliggøre ændringerne, giver jeg et komplet struktureksempel for vor gadget. Man kan lade denne gadget oprette i tilgift til de andre, hvis man skriver "dc.l gadget2" i det første lange ord af den sidste struktur.

```
gadget2:                    ;* struktur til en proportional-gadget
        dc.l    0           ;ikke flere gadgets
        dc.w    150,30      ;position
        dc.w    100,10      ;bredde og højde af rammen
        dc.w    4           ;flag: GADGIMAGE
        dc.w    2           ;activation: GADGIMMEDIATE
        dc.w    3           ;type: proportional gadget
        dc.l    mover       ;pointer på regulator-data
        dc.l    0           ;ingen select-struktur
        dc.l    0           ;ingen tekst
        dc.l    0           ;intet exclude
        dc.l    propinfo    ;pointer på propinfo-strukturen
        dc.w    3           ;gadget-ID
        dc.l    0           ;ingen User-data
```

*Bemærk to ejendommeligheder: der bliver peget på en image-struktur, der anvendes til skyderen, og indført en pointer på en yderligere struktur, som SpecialInfo-pointer.*

Først strukturen "mover", der udgør en image-struktur for skyderen. Her et eksempel på strukturen:

```
mover:                      ;* struktur til skyder-image
        dc.w    0,0         ;intet offset
        dc.w    16,7        ;16x12 pixels stort
        dc.w    1           ;et bit-plane
        dc.l    moverdata   ;pointer på image-data
        dc.b    1,0         ;farve: hvid
        dc.b    0           ;ingen fortsættelse
```

```
moverdata:                ;* image-data til skyder
    dc.w %0111111111111110
    dc.w %0101111111111010
    dc.w %0101011111101010
    dc.w %0101010110101010
    dc.w %0101011111101010
    dc.w %0101111111111010
    dc.w %0111111111111110
```

Hidtil har der ikke været noget nyt i det. Men nu til PropInfo-strukturen, der beskriver propotional-gadget"ens særlige egenskaber:

Strukturen begynder med et flag-ord, der indeholder følgende flag-bits:

| Bit | Værdi | Navn | Betydning: melding ved |
|---|---|---|---|
| 0 | 1 | AUTOKNOB | skyder oprettes automatisk |
| 1 | 2 | FREEHORIZ | tillader horisontal forskydning |
| 2 | 4 | FREEVERT | tillader vertikal forskydning |
| 3 | 8 | PROPBORDERLESS | frakobler automatiske indramning |
| 8 | $100 | KNOBHIT | sættes, når skyderen berøres |

De første fire bits kan sættes, for at indstille den repræsentation, man ønsker. Bit 8 sættes af Intuition, når man klikker ind på skyderen med musepointeren.

Bit 0, AUTOKNOB, tillader den simpleste form for proportional-gadget. Hvis denne bit sættes, så modtages ikke længere data om skyder-image't. I stedet for genereres en hvid skyder, der i sine mål er tilpasset til boxen, og den repræsenterede værdi. Et eksempel på tilpasningen:

Når der i et tekstprogram er tilladt en vertikal forskydning, og de viste linier af en lang tekst indstilles med skyderen, er en bestemt procentdel af teksten synlig i vinduet. Forholdet mellem det samlede linietal, og de viste tegn, udtrykkes med en AUTOKNOB som forholdet mellem rammen og skyderen. Jo større denne procentdel er, desto større bliver skyderen indenfor rammen.

Vi ser bort fra den mulighed, fordi en simpel hvid knop ikke er køn. Vil man alligevel prøve den, skal man passe på, at pointeren på imagedata ikke sættes på nul, men på en fire ord lang buffer, hvori Intuition indfører følgende værdier. Bufferen ser således ud:

```
puffer:
    dc.w  0           ;x-position af skyderen i boxen
    dc.w  0           ;y-postion i boxen
    dc.w  0           ;bredde af skyderen
    dc.w  0           ;højde af skyderen
```

Tilbage til PropInfo-strukturen. Eftersom vi ikke anvender AUTOKNOB og kun tillader horisontal forskydning, indsætter vi et 2 som flag:

```
propinfo:
    dc.w  2           ;flag: FREEHORIZ
```

I de næste to ord indføres den horisontale (HorizPot) og vertikale (VertPot) position af skyderen. Her betyder værdien 0 til venstre, resp. op, og værdien $FFFF til højre, resp. ned. Den med musen indstillede værdi efter forskydningen, ligger et sted herimellem. Til at begynde med sætter vi værdierne på 0. Efter en forskydning kan man finde en ny værdi.

```
    dc.w  0,0         ;x/y-position af skyderen
```

Nu kommer to ord mere, som angiver størrelsen af AUTOKNOB'en eller skyderens trinvidde, hvormed den bevæger sig, når man klikker i boxen ved siden af. Disse ord hedder HorizBody for den horisontale, og VertBody for den vertikale forskydning.

Hvis man, som f.eks. ved indstillingen af farveblandingen i preferences-programmet, vil tillade 16 trin ialt under forskydningen, så skal man stille værdien på den maksimale position/16, altså $FFF/$10=$FFF. Når man klikker i boxen ved siden af skyderen, bevæger den sig så 1/16 af boxens samlede bredde. Ved anvendelsen af AUTOKNOB-optionen er denne knop 1/16 så bred som hele boxen. Vi viser et eksempel:

```
    dc.w  $ffff/16    ;horisontal trinvidde
    dc.w  0           ;ingen vertikal forskydning
```

Nu følger 6 ord, der bliver initialiseret og beskrevet af Intuition.

```
    dc.w  0           ;box-bredde
    dc.w  0           ;box-højde
    dc.w  0           ;absolutte trinvidde horisontalt
    dc.w  0           ;og vertikalt
    dc.w  0           ;venstre kant af box
    dc.w  0           ;øvre kant af box
```

Her er hele PropInfo-strukturen:

```
propinfo:
    dc.w  2          ;flag: FREEHORIZ
    dc.w  0,0        ;x/y-position af skyderen
    dc.w  $ffff/16   ;horisontal trinvidde: 1/16
    dc.w  0          ;ingen vertikal forskydning
    dc.w  0          ;box-bredde
    dc.w  0          ;box-højde
    dc.w  0          ;absolutte trinvidde horisontalt
    dc.w  0          ;og vertikalt
    dc.w  0          ;venstre kant af box
    dc.w  0          ;øvre kant af box
```

Når man har indtastet det, kan man starte og afprøve programmet. Den position, der er indstillet, kan man se med "q propinfo", efter at have afbrudt programmet

Derefter kan man prøve at tillade den vertikale forskydning, ved at sætte flag-ordet på 6, den vertikale trinvidde på $FFFF/10 og højden af gadget'en på f.eks. 80. Man bør også prøve anvendelsen af AUTOKNOB ved at indsætte et 7 i flagene.

# 8. Programeksempler

Vi har lært en bunke forskellige effekter, som vi kan fremkalde med maskinsprog. Der mangler bare nogle grundlæggende ting, som kan bruges som værktøj ved programmeringen. Denne mangel skal vi nu afhjælpe. Vi vil se på nogle programeksempler, der afslører mange af de myter, der har været nævnt i bogen. Eksemplerne kan anvendes til egne programmer, næsten uden ændring.

## 8.1. Omstilling til supervisor-modus

Som det er nævnt i kapitlet om MC68000-processoren, har den to forskellige måder at operere på: user - og supervisor-modus. Det er ofte nødvendigt, at stille frem og tilbage mellem disse modi, hvad ikke er så simpelt endda.

Problemet består i, at man ikke må tilgå statusregistret i user-modus. En tilgang hertil resulterer i en exception (Privilege Violation = privilegiekrænkelse), der sluttelig ender med den bekendte gurumeditation.

Hvordan kan man nu komme til supervisor-modus?

Det er ikke noget problem. Amigaens operativsystem, resp. EXEC-bilioteket indeholder en funktion, der klarer det. Funktionen hedder SuperState, og kræver ingen parametre. Vi kan kalde funktionen, med følgende linier:

```
ExecBase   = 4      ;EXEC-basisadresse
Superstate = -150   ;omstillings-funktion

    ...
    move.l  ExecBase,a6      ;EXEC-basisadresse til A6
    jsr     SuperState(a6)   ;omstilling til supervisor
    move.l  d0,savesp        ;gem returværdi

    ....
savesp:     blk.l 1          ;plads til SP-værdi
```

I register D0 returneres stack-pointerens (SP) tilstand, der blev sat ved kaldet. Den befinder sig også i register A7, men det register forandrer sig hele tiden. Grunden hertil er, at i supervisor-modus, og dermed SP, arbejder alle Amigaens interrupts, der er meget talrige på denne computer. Vi vender tilbage til disse interrupts. Efter kaldet anvendes stadigvæk user-stacken, og ikke supervisor-stacken. Herigennem har man stadig mulighed for at tilgå

den gamle user-stack. Man skal passe på, at user-stacken er stor nok til, at de omtalte interrupts har plads nok til deres data.

Det vi har fået returneret i register D0, må først gemmes, fordi vi får brug for værdien senere. Når det kommer til stykket, skal vi også tilbage til user-modus, hvortil der findes endnu en EXEC-funktion. Det er UserState-funktionen, der skal have n parameter: den SP-værdi som SuperState-funktionen gav.

Da vi har gemt dette i det lange ord fra "savesp", kan vi skrive:

```
UserState  = -156

...
move.l  ExecBase,a6      ;EXEC-basisadresse til A6
move.l  savesp,d0        ;overgiv gamle SP til D0
jsr     UserState(a6)    ;tilbage til user-modus
```

Vi er tilbage i user-modus og med samme tilstand af user-stack-pointeren (USP) som før. Af den grund kan man også skrive en funktion, der skal udføres i supervisor-modus, som en underrutine. Den begynder med at kalde SuperState, gemmer returværdien, udfører den ønskede funktion, kalder UserState, og ender med RTS. Hvis tilstanden af USP på dette sted afveg fra begyndelsestilstanden, så vil RTS-kommadoen ikke fungere korrekt, og computeren vil foretage et hop "ud i det blå", og vil sandsynligvis crashe. Men her går det.

Så dukker spørgsmålet op: hvordan klarer operativsystemet at omstille til supervisor-modus? Det er nemt at forklare. Det sker på denne måde:

I SuperState-funktionen forsøges en tilgang til statusregistret. Det bevirker, at der optræder en exception, og der kaldes en rutine, hvis adresse står i det lange ord fra $20. Det drejer sig om exception-vektoren for privilegiekrænkelse. Den rutine, der forgrenes til, bliver automatisk kaldt af supervisor-modus'en. Der afprøves, hvorfra denne exception blev udløst. Hvis rutinen finder ud af, at oprindelsen til exceptionen ligger i EXEC-bibliotekets SuperState-rutine, hvis adresse den kender, så er sagen klar. Der forgrenes tilbage til rutinen, uden at der atter omstilles til user-modus. Det var det hele.

# 8.2. Exception-programmering

De, i kapitlet om processoren, beskrevne exceptions byder på mange muligheder, for at styre Amigaens funktion ad specielle veje. Vi kan styre systemfejl over i særlige baner, og således måske endda overliste guruen.

Her er endnu engang listen over vektorer, der anvendes til at hoppe til de tilsvarende exceptions-rutiner:

| Nummer | Adresse | Anvendelse ved |
|---|---|---|
| 2 | $008 | Bus Error |
| 3 | $00C | Address Error |
| 4 | $010 | Illegal ordre |
| 5 | $014 | Division med nul |
| 6 | $018 | CHK-ordre |
| 7 | $01C | TRAPV-ordre |
| 8 | $020 | Privilegiekrænkelse |
| 9 | $024 | Trace |
| 10 | $028 | Axxx-kommandoemulering |
| 11 | $02C | Fxxx-kommandoemulering |
| | $030-$038 | Reserveret |
| 15 | $03C | Uinitialiseret interrupt |
| | $040-$05F | Reserveret |
| 24 | $060 | Uberettiget interrupt |
| 25-31 | $064-$083 | Level 1-7 interrupt |
| 32-47 | $080-$0BF | TRAP-ordrer |
| | $0C0-$0FF | Reserveret |
| 64-255 | $100-$3FF | User-interrupt vektorer |

Lad os som eksempel tage de TRAP-ordrer, der ikke anvendes i Amiga-operativsystemet. En TRAP-ordre indsættes i et maskinprogram sammen med et tal mellem 0 og 15, for at kalde en af de 16 mulige TRAP-rutiner. Når ordren TRAP #0 udføres, så forgrener processoren (i supervisor-modus) til den rutine, hvis adresse ligger fra $80 i lageret. Rutinen skal slutte med en RTE-ordre (return from exception).

Skriv nogle TRAP-rutiner, der demonstrerer anvendelsen af TRAPs. Dertil behøves tre programdele:

1. **Initialiseringen af TRAP-vektoren.**
2. **Trap-rutinen selv, der skal ende med RTE.**
3. **En test-rutine, hvori TRAP-ordren kaldes.**

Initialiseringen er nem:

```
init:
    move.l  #trap0,$80      ;sæt vektor for TRAP #0
    rts
```

Rutinen med navnet "trap0" skal også oprettes. Som eksempel tages det program fra hardware-kapitlet, der frembringer en pippelyd.

Vi omskriver blot rutinen lidt, idet vi sætter et RTE til sidst i stedet for RTS. Sletter den linie, hvor sløjfetælleren D0 lades til tone-forsinkelsen, og ændrer ventesløjfen, så den også kan arbejde med lange ord. På den måde kan vi udløse en pippelyd af vilkårlig længde, ved at lade registeret med en vilkårlig værdi og et efterfølgende TRAP #0.

```
;* pippelydsfrembringelse med TRAP #0 **

ctlw   = $dff096              ;DMA-control

c0thi  = $dff0a0              ;tabeladresse HI
c0tlo  = c0thi+2              ;tabeladresse LO
c0tl   = c0thi+4              ;tabellængde
c0per  = c0thi+6              ;afspilningshastighed
c0vol  = c0thi +8             ;lydstyrke

trap0:                        ;* frembring kort pippelyd
    move.l  #table,c0thi      ;tabelbegyndelse
    move    #4,c0tl           ;tabellængde
    move    #300,c0per        ;afspilningshastighed
    move    #40,c0vol         ;lydstyrke

    move    #$8201,ctlw       ;start DMA resp. lyd

loop:
    subq.l  #1,d0             ;tæller -1
    bne     loop              ;tæl ned til 0

still:
    move    #1,ctlw           ;og sluk for lyd
    rte                       ;slut på exception

table:                        ;sound-tabel
    dc.b -40,-70,-40,0,40,70,40,0
```

Man skal igen passe på, at tabellen "table" ligger i chip-RAM'en ($00000-$7FFFF), da sound-chip'en ellers ikke kan få fat i disse data!

Når man har indtastet det, kan man tilføje følgende rutine for at teste:

```
test:
       move.l  #$2ffff,d0          ;overgiv lydlængde til D0
       trap    #0                  ;udløs exception: pip
       rts
```

Nu assemblerer man det hele med "a" og to gange return (SEKA), og starter derefter initialiserings-rutinen med "j init". Endnu sker der ikke noget.

Så indtaster man "j test": der lyder en pippelyd af ca. et sekunds varighed.

*Pas på: hvis man ændrer et eller andet i programmet og assemblerer igen, kan adressen på "trap0"-rutinen ændre sig. Før man lader TRAP-ordren udføre igen, skal man ubetinget gentage initialiseringen, for at computeren ikke forgrener på et forkert sted.*

# Appendix

## Biblioteksfunktioner i resume

Den følgende tabel giver et resume, af alle til rådighed stående biblioteker med deres funktioner. Tabellen benytter altid navnet på det bibliotek, hvori de følgende funktioner ligger, som overskrift.

Funktionerne er alle forsynet med deres negative offset, hexadecimalt og decimalt, deres navn og til sidst kort deres overgivelsesparametre. Parameter-navnene står i parenteser bagved funktionsnavnene; den anden parentes indeholder i samme rækkefølge de registre, hvori parametrene skal overgives. Hvis der ikke kræves nogen parametre, angives det med ().

### clist.library

```
-$001E   -30   InitCLPool (cLPool, size)(A0,D0)
-$0024   -36   AllocCList (cLPool)(A1)
-$002A   -42   FreeCList (cList)(A0)
-$0030   -48   FlushCList (cList)(A0)
-$0036   -54   SizeCList (cList)(A0)
-$003C   -60   PutCLChar (cList,byte)(A0,D0)
-$0042   -66   GetCLChar (cList)(A0)
-$0048   -72   UnGetCLChar (cList,byte)(A0,D0)
-$004E   -78   UnPutCLChar (cList)(A0)
-$0054   -84   PutCLWord (cList,word)(A0,D0)
-$005A   -90   GetCLWord (cList)(A0)
-$0060   -96   UnGetCLWord (cList,word)(A0,D0)
-$0066  -102   UnPutCLWord (cList)(A0)
-$006C  -108   PutCLBuf (cList,buffer,length)(A0,A1,D1)
-$0072  -114   GetCLBuf (cList,buffer,maxLength)(A0,A1,D1)
-$0078  -120   MarkCList (cList,offset)(A0,D0)
-$007E  -126   IncrCLMark (cList)(A0)
-$0084  -132   PeekCLMark (cList)(A0)
-$008A  -138   SplitCList (cList)(A0)
-$0090  -144   CopyCList (cList)(A0)
-$0096  -150   SubCList (cList,index,length)(A0,D0,D1)
-$009C  -156   ConcatCList (sourceCList,destCList)(A0,A1)
```

### console.library

```
-$002A   -42   CDInputHandler (events,device)(A0,A1)
-$0030   -48   RawKeyConvert (events,buffer,length,keyMap) (A0,A1,D1,A2)
```

## diskfont.library

| | | |
|---|---|---|
| -$001E | -30 | OpenDiskFont (textAttr)(A0) |
| -$0024 | -36 | AvailFonts (buffer,bufBytes,flags)(A0,D0,D1) |

## dos.library

| | | |
|---|---|---|
| -$001E | -30 | Open (name,accessMode)(D1,D2) |
| -$0024 | -36 | Close (file)(D1) |
| -$002A | -42 | Read (file,buffer,length)(D1,D2,D3) |
| -$0030 | -48 | Write (file,buffer,length)(D1,D2,D3) |
| -$0036 | -54 | Input () |
| -$003C | -60 | Output () |
| -$0042 | -66 | Seek (file,position,offset)(D1,D2,D3) |
| -$0048 | -72 | DeleteFile (name)(D1) |
| -$004E | -78 | Rename (oldName,newName)(D1,D2) |
| -$0054 | -84 | Lock (name,type)(D1,D2) |
| -$005A | -90 | UnLock (lock)(D1) |
| -$0060 | -96 | DupLock (lock)(D1) |
| -$0066 | -102 | Examine (lock,fileInfoBlock)(D1,D2) |
| -$006C | -108 | ExNext (lock,fileInfoBlock)(D1,D2) |
| -$0072 | -114 | Info (lock,parameterBlock)(D1,D2) |
| -$0078 | -120 | CreateDir (name)(D1) |
| -$007E | -126 | CurrentDir (lock)(D1) |
| -$0084 | -132 | IoErr () |
| -$008A | -138 | CreateProc (name,pri,segList,stackSize)(D1,D2,D3,D4) |
| -$0090 | -144 | Exit (returnCode)(D1) |
| -$0096 | -150 | LoadSeg (fileName)(D1) |
| -$009C | -156 | UnLoadSeg (segment)(D1) |
| -$00A2 | -162 | GetPacket (wait)(D1) |
| -$00A8 | -168 | QueuePacket (packet)(D1) |
| -$00AE | -174 | DeviceProc (name)(D1) |
| -$00B4 | -180 | SetComment (name,comment)(D1,D2) |
| -$00BA | -186 | SetProtection (name,mask)(D1,D2) |
| -$00C0 | -192 | DateStamp (date)(D1) |
| -$00C6 | -198 | Delay (timeout)(D1) |
| -$00CC | -204 | WaitForChar (file,timeout)(D1,D2) |
| -$00D2 | -210 | ParentDir (lock)(D1) |
| -$00D8 | -216 | IsInteractive (file)(D1) |
| -$00DE | -222 | Execute (string,file,file)(D1,D2,D3) |

## exec.library

| | | |
|---|---|---|
| -$001E | -30 | Supervisor () |
| -$0024 | -36 | ExitIntr () |

| | | |
|---|---|---|
| -$002A | -42 | Schedule () |
| -$0030 | -48 | Reschedule () |
| -$0036 | -54 | Switch () |
| -$003C | -60 | Dispatch () |
| -$0042 | -66 | Exception () |
| -$0048 | -72 | InitCode (startClass,version)(D0,D1) |
| -$004E | -78 | InitStruct (initTable,memory,size)(A1,A2,D0) |
| -$0054 | -84 | MakeLibrary (funcInit,structInit,libInit,dataSize, codeSize) (A0,A1,A2,D0,D1) |
| -$005A | -90 | MakeFunctions (target,functionArray,funcDispBase) (A0,A1,A2) |
| -$0060 | -96 | FindResident (name)(A1) |
| -$0066 | -102 | InitResident (resident,segList)(A1,D1) |
| -$006C | -108 | Alert (alertNum,parameters)(D7,A5) |
| -$0072 | -114 | Debug () |
| -$0078 | -120 | Disable () |
| -$007E | -126 | Enable () |
| -$0084 | -132 | Forbid () |
| -$008A | -138 | Permit () |
| -$0090 | -144 | SetSR (newSR,mask)(D0,D1) |
| -$0096 | -150 | SuperState () |
| -$009C | -156 | UserState (sysStack)(D0) |
| -$00A2 | -162 | SetIntVector (intNumber,interrupt)(D0,A1) |
| -$00A8 | -168 | AddIntServer (intNumber,interrupt)(D0,A1) |
| -$00AE | -174 | RemIntServer (intNumber,interrupt)(D0,A1) |
| -$00B4 | -180 | Cause (interrupt)(A1) |
| -$00BA | -186 | Allocate (freeList,byteSize)(A0,D0) |
| -$00C0 | -192 | Deallocate (freeList,memoryBlock,byteSize)(A0,A1,D0) |
| -$00C6 | -198 | AllocMem (byteSize,requirements)(D0,D1) |
| -$00CC | -204 | AllocAbs (byteSize,location)(D0,A1) |
| -$00D2 | -210 | FreeMem (memoryBlock,byteSize)(A1,D0) |
| -$00D8 | -216 | AvailMem (requirements)(D1) |
| -$00DE | -222 | AllocEntry (entry)(A0) |
| -$00E4 | -228 | FreeEntry (entry)(A0) |
| -$00EA | -234 | Insert (list,node,pred)(A0,A1,A2) |
| -$00F0 | -240 | AddHead (list,node)(A0,A1) |
| -$00F6 | -246 | AddTail (list,node)(A0,A1) |
| -$00FC | -252 | Remove (node)(A1) |
| -$0102 | -258 | RemHead (list)(A0) |
| -$0108 | -264 | RemTail (list)(A0) |
| -$010E | -270 | Enqueue (list,node)(A0,A1) |
| -$0114 | -276 | FindName (list,name)(A0,A1) |
| -$011A | -282 | AddTask (task,initPC,finalPC)(A1,A2,A3) |
| -$0120 | -288 | RemTask (task)(A1) |
| -$0126 | -294 | FindTask (name)(A1) |
| -$012C | -300 | SetTaskPri (task,priority)(A1,D0) |
| -$0132 | -306 | SetSignal (newSignals,signalSet)(D0,D1) |
| -$0138 | -312 | SetExcept (newSignals,signalSet)(D0,D1) |

```
-$013E  -318  Wait (signalSet)(D0)
-$0144  -324  Signal (task,signalSet)(A1,D0)
-$014A  -330  AllocSignal (signalNum)(D0)
-$0150  -336  FreeSignal (signalNum)(D0)
-$0156  -342  AllocTrap (trapNum)(D0)
-$015C  -348  FreeTrap (trapNum)(D0)
-$0162  -354  AddPort (port)(A1)
-$0168  -360  RemPort (port)(A1)
-$016E  -366  PutMsg (port,message)(A0,A1)
-$0174  -372  GetMsg (port)(A0)
-$017A  -378  ReplyMsg (message)(A1)
-$0180  -384  WaitPort (port)(A0)
-$0186  -390  FindPort (name)(A1)
-$018C  -396  AddLibrary (library)(A1)
-$0192  -402  RemLibrary (library)(A1)
-$0198  -408  OldOpenLibrary (libName)(A1)
-$019E  -414  CloseLibrary (library)(A1)
-$01A4  -420  SetFunction (library,funcOffset,funcEntry)(A1,A0,D0)
-$01AA  -426  SumLibrary (library)(A1)
-$01B0  -432  AddDevice (device)(A1)
-$01B6  -438  RemDevice (device)(A1)
-$01BC  -444  OpenDevice (devName,unit,ioRequest,flags)(A0,D0,A1,D1)
-$01C2  -450  CloseDevice (ioRequest)(A1)
-$01C8  -456  DoIO (ioRequest)(A1)
-$01CE  -462  SendIO (ioRequest)(A1)
-$01D4  -468  CheckIO (ioRequest)(A1)
-$01DA  -474  WaitIO (ioRequest)(A1)
-$01E0  -480  AbortIO (ioRequest)(A1)
-$01E6  -486  AddRescource (rescource)(A1)
-$01EC  -492  RemRescource (rescource)(A1)
-$01F2  -498  OpenRescource (resName,version)(A1,D0)
-$01F8  -504  RawIOInit ()
-$01FE  -510  RawMayGetChar ()
-$0204  -516  RawPutChar (char)(D0)
-$020A  -522  RawDoFmt ()(A0,A1,A2,A3)
-$0210  -528  GetCC ()
-$0216  -534  TypeOfMem (address)(A1)
-$021C  -540  Procedure (semaport,bidMsg)(A0,A1)
-$0222  -546  Vacate (semaport)(A0)
-$0228  -552  OpenLibrary (libName,version)(A1,D0)
```

## graphics.library

```
-$001E   -30  BltBitMap (srcBitMap,srcX,srcY,destBitMap,destX,destY,
              sizeX,sizeY,minterm,mask,tempA) (A0,D0,D1,A1,D2,D3,D4,
              D5,D6,D7,A2)
-$0024   -36  BltTemplate (source,srcX,srcMod,destRastPort,destX,destY,
```

```
                sizeX,sizeY)(A0,D0,D1,A1,D2,D3,D4,D5)
-$002A   -42    ClearEOL (rastPort)(A1)
-$0030   -48    ClearScreen (rastPort)(A1)
-$0036   -54    TextLength (RastPort,string,count)(A1,A0,D0)
-$003C   -60    Text (RastPort,String,count)(A1,A0,D0)
-$0042   -66    SetFont (RAstPortID,textFont)(A1,A0)
-$0048   -72    OpenFont (textAttr)(A0)
-$004E   -78    CloseFont (textFont)(A1)
-$0054   -84    AskSoftStyle (rastPort)(A1)
-$005A   -90    SetSoftStyle (rastPort,style,enable)(A1,D0,D1)
-$0060   -96    AddBob (bob,rastPort)(A0,A1)
-$0066   -102   AddVSprite (vSprite,rastPort)(A0,A1)
-$006C   -108   DoCollision (rastPort)(A1)
-$0072   -114   DrawGList (rastPort,viewPort)(A1,A0)
-$0078   -120   InitGels (dummyHead,dummyTail,GelsInfo)(A0,A1,A2)
-$007E   -126   InitMasks (vSprite)(A0)
-$0084   -132   RemIBob (bob,rastPort,viewPort)(A0,A1,A2)
-$008A   -138   RemVSprite (vSprite)(A0)
-$0090   -144   SetCollision (type,routine,gelsInfo)(D0,A0,A1)
-$0096   -150   SortGList (rastPort)(A1)
-$009C   -156   AddAnimObj (obj,animationKey,rastPort)(A0,A1,A2)
-$00A2   -162   Animate (animationKey,rastPort)(A0,A1)
-$00A8   -168   GetGBuffers (animationObj,rastPort,doubleBuffer)
                (A0,A1,D0)
-$00AE   -174   InitGMasks (animationObj)(A0)
-$00B4   -180   GelsFuncE ()
-$00BA   -186   GelsFuncF ()
-$00C0   -192   LoadRGB4 (viewPort,colors,count)(A0,A1,D0)
-$00C6   -198   InitRastPort (rastPort)(A1)
-$00CC   -204   InitVPort (viewPort)(A0)
-$00D2   -210   MrgCop (view)(A1)
-$00D8   -216   MakeVPort (view,viewPort)(A0,A1)
-$00DE   -222   LoadView (view)(A1)
-$00E4   -228   WaitBlit ()
-$00EA   -234   SetRast (rastPort,color)(A1,D0)
-$00F0   -240   Move (rastPort,x,y)(A1,D0,D1)
-$00F6   -246   Draw (rastPort,x,y)(A1,D0,D1)
-$00FC   -252   AreaMove (rastPort,x,y)(A1,D0,D1)
-$0102   -258   AreaDraw (rastPort,x,y)(A1,D0,D1)
-$0108   -264   AreaEnd (rastPort)(A1)
-$010E   -270   WaitTOF ()
-$0114   -276   QBlit (blit)(A1)
-$011A   -282   InitArea (areaInfo,vectorTable,vectorTableSize)(A0,A1,D0)
-$0120   -288   SetRGB4 (viewPort,index,r,g,b)(A0,D0,D1,D2,D3)
-$0126   -294   QBSBlit (blit)(A1)
-$012C   -300   BltClear (memory,size,flags)(A1,D0,D1)
-$0132   -306   RectFill (rastPort,xl,yl,xu,yu)(A1,D0,D1,D2,D3)
-$0138   -312   BltPattern (rastPort,ras,xl,yl,maxX,maxY,fillBytes)
```

```
              (A1,A0,D0,D1,D2,D3,D4)
-$013E  -318  ReadPixel (rastPort,x,y)(A1,D0,D1)
-$0144  -324  WritePixel (rastPort,x,y)(A1,D0,D1)
-$014A  -330  Flood (rastPort,mode,x,y)(A1,D2,D0,D1)
-$0150  -336  PolyDraw (rastPort,count,polyTable)(A1,D0,A0)
-$0156  -342  SetAPen (rastPort,pen)(A1,D0)
-$015C  -348  SetBPen (rastPort,pen)(A1,D0)
-$0162  -354  SetDrMd (rastPort,drawMode)(A1,D0)
-$0168  -360  InitView (view)(A1)
-$016E  -366  CBump (copperList)(A1)
-$0174  -372  CMove (copperList,destination,data)(A1,D0,D1)
-$017A  -378  CWait (copperList,x,y)(A1,D0,D1)
-$0180  -384  VBeamPos ()
-$0186  -390  InitBitMap (bitMap,depth,width,heigth)(A0,D0,D1,D2)
-$018C  -396  ScrollRaster (rastPort,dX,dY,minx,miny,maxx,maxy)(A1,D0,
              D1,D2,D3,D4,D5)
-$0192  -402  WaitBOVP (viewPort)(A0)
-$0198  -408  GetSprite (simpleSprite,num)(A0,D0)
-$019E  -414  FreeSprite (num)(D0)
-$01A4  -420  ChangeSprite (vp,simpleSprite,data)(A0,A1,A2)
-$01AA  -426  MoveSprite (viewPort,simpleSprite,x,y)(A0,A1,D0,D1)
-$01B0  -432  LockLayerRom (layer)(A5)
-$01B6  -438  UnlockLayerRom (layer)(A5)
-$01BC  -444  SyncSBitMap (1)(A0)
-$01C2  -450  CopySBitMap (11,12)(A0,A1)
-$01C8  -456  OwnBlitter ()
-$01CE  -462  DisownBlitter ()
-$01D4  -468  InitTmpRas (tmpras,buff,size)(A0,A1,D0)
-$01DA  -474  AskFont (rastPort,textAttr)(A1,A0)
-$01E0  -480  AddFont (textFont)(A1)
-$01E6  -486  RemFont (textFont)(A1)
-$01EC  -492  AllocRaster (width,heigth)(D0,D1)
-$01F2  -498  FreeRaster (planeptr,width,heigth)(A0,D0,D1)
-$01F8  -504  AndRectRegion (rgn,rect)(A0,A1)
-$01FE  -510  OrRectRegion (rgn,rect)(A0,A1)
-$0204  -516  NewRegion ()
-$020A  -522  ** reserviert **
-$0210  -528  ClearRegion (rgn)(A0)
-$0216  -534  DisposeRegion (rgn)(A0)
-$021C  -540  FreeVPortCopLists (viewPort)(A0)
-$0222  -546  FreeCopList (coplist)(A0)
-$0228  -552  ClipBlit (srcrp,srcX,srcY,destrp,destX,destY,sizeX,
              sizeY,minterm)(A0,D0,D1,A1,D2,D3,D4,D5,D6)
-$022E  -558  XorRectRegion (rgn,rect)(A0,A1)
-$0234  -564  FreeCprList (cprlist)(A0)
-$023A  -570  GetColorMap (entries)(D0)
-$0240  -576  FreeColorMap (colormap)(A0)
-$0246  -582  GetRGB4 (colormap,entry)(A0,D0)
```

| | | |
|---|---|---|
| -$024C | -588 | ScrollVPort (vp)(A0) |
| -$0252 | -594 | UCopperListInit (copperlist,num)(A0,D0) |
| -$0258 | -600 | FreeGBuffers (animationObj,rastPort, doubleBuffer)(A0,A1,D0) |
| -$025E | -606 | BltBitMapRastPort (srcbm,srcx,srcy,destrp,destX,destY, sizeX,sizeY,minter)(A0,D0,D1,A1,D2,D3,D4,D5,D6) |

## icon.library

| | | |
|---|---|---|
| -$001E | -30 | GetWBObject (name)(A0) |
| -$0024 | -36 | PutWBObject (name,object)(A0,A1) |
| -$002A | -42 | GetIcon (name,icon,freelist)(A0,A1,A2) |
| -$0030 | -48 | PutIcon (name,icon)(A0,A1) |
| -$0036 | -54 | FreeFreeList (freelist)(A0) |
| -$003C | -60 | FreeWBObject (WBObject)(A0) |
| -$0042 | -66 | AllocWBObject () |
| -$0048 | -72 | AddFreeList (freelist,mem,size)(A0,A1,A2) |
| -$004E | -78 | GetDiskObject (name)(A0) |
| -$0054 | -84 | PutDiskObject (name,diskobj)(A0,A1) |
| -$005A | -90 | FreeDiskObj (diskobj)(A0) |
| -$0060 | -96 | FindToolType (toolTypeArray,typeName)(A0,A1) |
| -$0066 | -102 | MatchToolValue (typeString,value)(A0,A1) |
| -$006C | -108 | BumbRevision (newname,oldname)(A0,A1) |

## intuition.library

| | | |
|---|---|---|
| -$001E | -30 | OpenIntuition () |
| -$0024 | -36 | Intuition (ievent)(A0) |
| -$002A | -42 | AddGadget (AddPtr,Gadget,Position)(A0,A1,D0) |
| -$0030 | -48 | ClearDMRequest (Window)(A0) |
| -$0036 | -54 | ClearMenuStrip (Window)(A0) |
| -$003C | -60 | ClearPointer (Window)(A0) |
| -$0042 | -66 | CloseScreen (Screen)(A0) |
| -$0048 | -72 | CloseWindow (Window)(A0) |
| -$004E | -78 | CloseWorkBench () |
| -$0054 | -84 | CurrentTime (Seconds,Micros)(A0,A1) |
| -$005A | -90 | DisplayAlert (AlertNumber,String,Height)(D0,A0.D1) |
| -$0060 | -96 | DisplayBeep (Screen)(A0) |
| -$0066 | -102 | DoubleClick (sseconds,smicros,cseconds,cmicros) (D0,D1,D2,D3) |
| -$006C | -108 | DrawBorder (Rport,Border,LeftOffset,TopOffset) (A0,A1,D0,D1) |
| -$0072 | -114 | DrawImage (RPort,Image,LeftOffset,TopOffset)(A0,A1,D0,D1) |
| -$0078 | -120 | EndRequest (requester,window)(A0,A1) |

```
-$007E  -126  GetDefPrefs (preferences,size)(A0,D0)
-$0084  -132  GetPrefs (preferences,size)(A0,D0)
-$008A  -138  InitRequester (req)(A0)
-$0090  -144  ItemAddress (MenuStrip,MenuNumber)(A0,D0)
-$0096  -150  ModifyIDCMP (Window,Flags)(A0,D0)
-$009C  -156  ModifyProp (Gadget,Ptr,Reg,Flags,HPos,VPos,HBody,VBody)
              (A0,A1,A2,D0,D1,D2,D3,D4)
-$00A2  -162  MoveScreen (Screen,dx,dy)(A0,D0,D1)
-$00A8  -168  MoveWindow (Window,dx,dy)(A0,D0,D1)
-$00AE  -174  OffGadget (Gadget,Ptr,Req)(A0,A1,A2)
-$00B4  -180  OffMenu (Window,MenuNumber)(A0,D0)
-$00BA  -186  OnGadget (Gadget,Ptr,Req)(A0,A1,A2)
-$00C0  -192  OnMenu (Window,MenuNumber)(A0,D0)
-$00C6  -198  OpenScreen (OSArgs)(A0)
-$00CC  -204  OpenWindow (OWArgs)(A0)
-$00D2  -210  OpenWorkBench ()
-$00D8  -216  PrintIText (rp,itext,left,top)(A0,A1,D0,D1)
-$00DE  -222  RefreshGadgets (Gadgets,Ptr,Req)(A0,A1,A2)
-$00E4  -228  RemoveGadgets (RemPtr,Gadget)(A0,A1)
-$00EA  -234  ReportMouse (Window,Boolean)(A0,D0)
-$00F0  -240  Request (Requester,Window)(A0,A1)
-$00F6  -246  ScreenToBack (Screen)(A0)
-$00FC  -252  SCreenToFront (Screen)(A0)
-$0102  -258  SetDMRequest (Window,req)(A0,A1)
-$0108  -264  SetMenuStrip (Window,Menu)(A0,A1)
-$010E  -270  SetPointer (Window,Pointer,Height,Width,XOffset,YOffset)
              (A0,A1,D0,D1,D2,D3)
-$0114  -276  SetWindowTitles (Window,windowTitle,screenTitle)
              (A0,A1,A2)
-$011A  -282  ShowTitle (Screen,ShowIt)(A0,D0)
-$0120  -288  SizeWindow (Window,dx,dy)(A0,D0,D1)
-$0126  -294  ViewAddress ()
-$012C  -300  ViewPortAddress (Window)(A0)
-$0132  -306  WindowToBack (Window)(A0)
-$0138  -312  WindowToFront (Window)(A0)
-$013E  -318  WindowLimits (Window,minwidth,minheight,maxwidth,
              maxheight)(A0,D0,D1,D2,D3)
-$0144  -324  SetPrefs (preferences,size,flag)(A0,D0,D1)
-$014A  -330  IntuiTextLength (itext)(A0)
-$0150  -336  WBenchToBack ()
-$0156  -342  WBenchToFront ()
-$015C  -348  AutoRequest (WIndow,Body,PText,NText,PFlag,NFlag,W,H)
              (A0,A1,A2,A3,D0,D1,D2,D3)
-$0162  -354  BeginRefresh (Window)(A0)
-$0168  -360  BuildSysRequest (Window,Body,PosText,NegText,Flags,W,H)
              (A0,A1,A2,A3,D0,D1,D2)
-$016E  -366  EndRefresh (Window,Complete)(A0,D0)
-$0174  -372  FreeSysRequest (Window)(A0)
```

| | | |
|---|---|---|
| -$017A | -378 | MakeScreen (Screen)(A0) |
| -$0180 | -384 | RemakeDisplay () |
| -$0186 | -390 | RethinkDisplay () |
| -$018C | -396 | AllocRemember (RememberKey,Size,Flags)(A0,D0,D1) |
| -$0192 | -402 | AlohaWorkbench (wbport)(A0) |
| -$0198 | -408 | FreeRemember (RememberKey,ReallyForget)(A0,D0) |
| -$019E | -414 | LockIBase (dontknow)(D0) |
| -$01A4 | -420 | UnlockIBase (IBLock)(A0) |

## layers.library

| | | |
|---|---|---|
| -$001E | -30 | InitLayers (li)(A0) |
| -$0024 | -36 | CreateUpfrontLayer (li,bm,x0,y0,x1,y1,flags,bm2)(A0,A1, D0,D1,D2,D3,D4,A2) |
| -$002A | -42 | CreateBehindLayer (li,bm,x0,y0,x1,y1,flags,bm2)(A0,A1,D0, D1,D2,D3,D4,A2) |
| -$0030 | -48 | UpfrontLayer (li,layer)(A0,A1) |
| -$0036 | -54 | BehindLayer (li,layer)(A0,A1) |
| -$003C | -60 | MoveLayer (li,layer,dx,dy)(A0,A1,D0,D1) |
| -$0042 | -66 | SizeLayer (li,layer,dx,dy)(A0,A1,D0,D1) |
| -$0048 | -72 | ScrollLayer (li,layer,dx,dy)(A0,A1,D0,D1) |
| -$004E | -78 | BeginUpdate (layer)(A0) |
| -$0054 | -84 | EndUpdate (layer)(A0) |
| -$005A | -90 | DeleteLayer (li,layer)(A0,A1) |
| -$0060 | -96 | LockLayer (li,layer)(A0,A1) |
| -$0066 | -102 | UnlockLayer (li,layer)(A0,A1) |
| -$006C | -108 | LockLayers (li)(A0) |
| -$0072 | -114 | UnlockLayers (li)(A0) |
| -$0078 | -120 | LockLayerInfo (li)(A0) |
| -$007E | -126 | SwapBitsRastPortClipRect (rp,cr)(A0,A1) |
| -$0084 | -132 | WhichLayer (li,x,y)(A0,D0,D1) |
| -$008A | -138 | UnlockLayerInfo (li)(A0) |
| -$0090 | -144 | NewLayerInfo () |
| -$0096 | -150 | DisposeLayerInfo (li)(A0) |
| -$009C | -156 | FattenLayerInfo (li)(A0) |
| -$00A2 | -162 | ThinLayerInfo (li)(A0) |
| -$00A8 | -168 | MoveLayerInFrontOf (layer_to_move,layer_to_be_in_front_ of)(A0,A1) |

## mathffp.library

| | | |
|---|---|---|
| -$001E | -30 | SPFix (float)(D0) |
| -$0024 | -36 | SPFlt (integer)(D0) |
| -$002A | -42 | SPCmp (leftFloat,rightFloat)(D1,D0) |
| -$0030 | -48 | SPTst (float)(D1) |

| | | |
|---|---|---|
| -$0036 | -54 | SPAbs (float)(D0) |
| -$003C | -60 | SPNeg (float)(D0) |
| -$0042 | -66 | SPAdd (leftFloat,rightFloat)(D1,D0) |
| -$0048 | -72 | SPSub (leftFloat,rightFloat)(D1,D0) |
| -$004E | -78 | SPMul (leftFloat,rightFloat)(D1,D0) |
| -$0054 | -84 | SPDiv (leftFloat,rightFloat)(D1,D0) |

## mathieeedoubbas.library

| | | |
|---|---|---|
| -$001E | -30 | IEEEDPFix (integer,integer)(D0,D1) |
| -$0024 | -36 | IEEEDPFlt (integer)(D0) |
| -$002A | -42 | IEEEDPCmp (integer,integer,integer,integer)(D0,D1,D2,D3) |
| -$0030 | -48 | IEEEDPTst (integer,integer)(D0,D1) |
| -$0036 | -54 | IEEEDPAbs (integer,integer)(D0,D1) |
| -$003C | -60 | IEEEDPNeg (integer,integer)(D0,D1) |
| -$0042 | -66 | IEEEDPAdd (integer,integer,integer,integer)(D0,D1,D2,D3) |
| -$0048 | -72 | IEEEDPSub (integer,integer,integer,integer)(D0,D1,D2,D3) |
| -$004E | -78 | IEEEDPMul (integer,integer,integer,integer)(D0,D1,D2,D3) |
| -$0054 | -84 | IEEEDPDiv (integer,integer,integer,integer)(D0,D1,D2,D3) |

## mathtrans.library

| | | |
|---|---|---|
| -$001E | -30 | SPAtan (float)(D0) |
| -$0024 | -36 | SPSin (float)(D0) |
| -$002A | -42 | SPCos (float)(D0) |
| -$0030 | -48 | SPTan (float)(D0) |
| -$0036 | -54 | SPSincos (leftFloat,rightFloat)(D1,D0) |
| -$003C | -60 | SPSinh (float)(D0) |
| -$0042 | -66 | SPCosh (float)(D0) |
| -$0048 | -72 | SPTanh (float)(D0) |
| -$004E | -78 | SPExp (float)(D0) |
| -$0054 | -84 | SPLog (float)(D0) |
| -$005A | -90 | SPPow (leftFloat,rightFloat)(D1,D0) |
| -$0060 | -96 | SPSqrt (float)(D0) |
| -$0066 | -102 | SPTieee (float)(D0) |
| -$006C | -108 | SPFieee (float)(D0) |
| -$0072 | -114 | SPAsin (float)(D0) |
| -$0078 | -120 | SPAcos (float)(D0) |
| -$007E | -126 | SPLog10 (float)(D0) |

## potgo.library

| | | |
|---|---|---|
| -$0006 | -6 | AllocPotBits (bits)(D0) |

```
-$000C   -12   FreePotBits (bits)(D0)
-$0012   -18   WritePotgo (word,mask)(D0,D1)
```

## timer.library

```
-$002A   -42   AddTime (dest,src)(A0,A1)
-$0030   -48   SubTime (dest,src)(A0,A1)
-$0036   -54   CmpTime (dest,src)(A0,A1)
```

## translator.library

```
-$001E   -30   Translate (inputString,inputLength,outputBuffer,
               bufferSize)(A0,D0,A1,D1)
```

## MC 68000-ordre oversigt

Anvendte forkortelser eller symboler:

| | |
|---|---|
| **label** | en label eller en adresse |
| **reg** | register |
| **An** | adresseregister n |
| **Dn** | dataregister n |
| **kilde** | kildeoperand |
| **mål** | måloperand |
| **<ea>** | adresse eller register |
| **#n** | direkte værdi |

| Mnemonic | | Betydning |
|---|---|---|
| **ABCD** | kilde,mål | Addition af to BCD-tal. |
| **ADD** | kilde,mål | Binær addition. |
| **ADDA** | kilde,an | Binær addition til et adresseregister. |
| **ADDI** | #n,<ea> | Addition med en konstant. |
| **ADDQ** | #n,<ea> | Hurtig addition af en konstant, der kun må være mellem 0 og 7. |
| **ADDX** | kilde,mål | Addition med overførelse til x-flag. |
| **AND** | kilde,mål | Logisk OG. |
| **ANDI** | #n,<ea> | Logisk OG med en konstant. |
| **ASL** | n,<ea> | Aritmetisk forskydning mod venstre (*2↑n). |
| **ASR** | n,<ea> | Aritmetisk forskydning mod højre (/2↑n). |
| **Bcc** | label | Forgren betinget, alt efter betingelse. |
| **BCHG** | #n,<ea> | Forandrer bit n (nul bliver til 1 og omvendt). |
| **BCLR** | #n,<ea> | Slet bit n. |
| **BRA** | label | Forgren ubetinget (ligesom JMP). |
| **BSET** | #n,<ea> | Sæt bit n. |
| **BSR** | label | Forgren til et underprogram. Ligesom med JSR bliver returadressen herunder lagt i stacken og returneret ved RTS |
| **BTST** | #n,<ea> | Test bit n, resultatet står i z-flaget. |
| **CHK** | <ea>,Dx | Afprøv grænserne for et dataregister, udløs evt. CHK-instruction exception'en. |
| **CLR** | <ea> | Slet en operand. |
| **CMP** | kilde,mål | Sammenligning af to operander. |
| **CMPA** | <ea>,An | Sammenligning med et adresseregister. |
| **CMPI** | #n,<ea> | Sammenligning med en konstant. |
| **CMPM** | kilde,maal | Sammenligning af to lageroperander. |
| **DBcc** | reg.label | Afprøv betingelse, dekrementer og forgren. Denne ordre anvendes hyppigt til løkker. |
| **DIVS** | kilde,mål | Fortegnsrigtig division af 32-bit mål- med 16-bit kilde-operanden. Resultatet af divisionen ligger derefter i målets LO-ord, resten af divisionen i HI-ordet. |

| Mnemonic | | Betydning |
|---|---|---|
| **DIVU** | kilde,mål | Division uden fortegn, ellers som DIVS. |
| **EOR** | kilde,mål | Eksklusivt ELLER. |
| **EORI** | #n,<ea> | Eksklusivt ELLER med en konstant. |
| **EXG** | Rn,Rn | Udveksling af to registerindhold (ikke at forveksle med SWAP!). |
| **EXT** | Dn | Fortegnsrigtig udvidelse til dobbelt bredde. |
| **JMP** | label | Hop til adresse (som ved BRA). |
| **JSR** | label | Hop til et underprogram. Også her lægges returhopadressen i stacken, RTS fører tilbage. |
| **LEA** | <ea>,An | Indlæs en effektiv adresse i adresseregister An. |
| **LINK** | An,#n | Opbyg et stackområde. |
| **LSL** | n,<ea> | Logisk forskydning mod venstre. |
| **LSR** | n,<ea> | Logisk forskydning mod højre. |
| **MOVE** | kilde,mål | Overfør en værdi fra kilde til mål. |
| **MOVE** | SR,<ea> | Overfør statusregister-indholdet. |
| **MOVE** | <ea>,SR | Overfør statusregister-indholdet. |
| **MOVE** | <ea>,CCR | Load flag. |
| **MOVE** | USP,<ea> | Overfør user-stackpointeren. |
| **MOVE** | <ea>,USP | Overfør user-stackpointeren. |
| **MOVEA** | <ea>,An | Overfør en værdi til adresseregister An. |
| **MOVEM** | regs,<ea> | Overfør flere registre på en gang. |
| **MOVEM** | <ea>,regs | Overfør flere registre på en gang. |
| **MOVEP** | kilde,mål | Overfør data til periferi. |
| **MOVEQ** | #n,Dn | Overfør hurtigt en 8-bit konstant til dataregister Dn. |
| **MULS** | kilde,mål | Fortegnsrigtig multiplikation af to ord til et langt ord. |
| **MULU** | kilde,mål | Multiplikation uden fortegn, ellers som MULS. |
| **NBCD** | kilde,mål | Negation af et BCD-tal (ni-komplement). |
| **NEG** | <ea> | Negation af en operand (to-komplement). |
| **NEGX** | <ea> | Negation af en operand med overførelse. |

| **Mnemonic** | | **Betydning** |
|---|---|---|
| **NOP** | | Ingen funktion (no operation). |
| **NOT** | <ea> | Inversion af en operand. |
| **OR** | kilde,mål | Logisk ELLER. |
| **ORI** | #n,<ea> | Logisk ELLER med en konstant. |
| **PEA** | <ea> | Læg en adresse i stacken. |
| **RESET** | | Reset periferi (forsigtig!). |
| **ROL** | n,<ea> | Rotation til venstre. |
| **ROR** | n,<ea> | Rotation til højre. |
| **ROXL** | n,<ea> | Rotation til venstre med overførelse til x-flag. |
| **ROXR** | n,<ea> | Rotation til højre med overførelse til x-flag. |
| **RTE** | | Tilbagevenden fra en exception. |
| **RTR** | | Tilbagevenden med load'ning af flag. |
| **RTS** | | Tilbagevenden fra et underprogram (efter BSR eller JSR). |
| **SBCD** | kilde,mål | Subtraktion af to BCD-tal. |
| **Scc** | <ea> | Sæt en byte på -1, hvis betingelsen er opfyldt. |
| **STOP** | | Stands udførelsen (forsigtig!), eventuelt TRAPV-exception. |
| **SUB** | kilde,mål | Binær subtraktion. |
| **SUBA** | <ea>,An | Binær subtraktion af et adresseregister. |
| **SUBI** | #n,<ea> | Subtraktion af en konstant. |
| **SUBQ** | #n,<ea> | Hurtig subtraktion af en 3-bit konstant. |
| **SUBX** | kilde,maal | Subtraktion med overførelse til x-flaget. |
| **SWAP** | Dn | Ombyt registerhalvdelene (de øverste 16 bits med de nederste). |
| **TAS** | <ea> | Prøv en byte og sæt bit 7. |
| **TRAP** | #n | Hop til en exception. |
| **TRAPV** | | Prøv, om et overløbs-flag er sat, test så en operand med TST <ea> og sæt eventuelt N- og Z-flag. |
| **UNLK** | An | Nedlæg et stackområde. |

# Ordforklaring (ikke komplet)

Et skønsomt udvalg, som forlaget håber kan være til nytte for Dem.

## A.

**Acces:**
Den tid, der går fra data kaldes indtil de er tilgængelige. Kaldes også tilgangstid.

**ADD:**
Forkortelse for addition.

**Adresse:**
Herunder forstår man en betegnelse for en bestemt lagerposition eller et bestemt lagerområde.

**Algoritme:**
En fuldstændig beskrevet fremgangsmåde til løsning af et givet program.

**Arbejdslager:**
Det område af hukommelsen, der bruges til lagring og bearbejdning af data. kaldes også RAM-området.

**ASCII:**
En amerikansk standard for udveksling af bogstaver og symboler. ASCII er en kodetabel, hvori alle bogstaver, tal og specialtegn har fået en talkode. Denne talkode er international. Oprindelig udviklet til brug for telex-kommunikation.

**Assembler:**
Assembler er et program der oversætter fra assemblersprog til maskinsprog.

**Assemblersprog:**
Det programmeringssprog, kaldes for symbolsk maskinsprog og ligger meget op tæt ved maskinsprog og maskinkode.

**Asynkron Dataoverførsel:**
En arbejdsmetode, ved hvilken de enkelte arbejdsskridts tidsenheder enten bestemmes af hvert arbejdsskridts længde eller af uregelmæssige eksterne signaler, kaldes asynkron dataoverførsel.

**Attribut:**

Særlig egenskab eller kendetegn.

## B.

**BASIC:**

Beginners All-purpose Symbolic Instruction Code. Sikkert det mest udbredte og mest anvendte programmeringssprog på hjemmecomputere og minicomputere (PC-ere)

**Batchfilen:**

Batchfilen er en fil, bestående af data, af samme karakter, f.eks. udskrift af lønsedler kaldes for en batchkørsel (gruppekørsel).

**BAUD:**

Dette er enheden for dataoverførslens skridthastighed og svarer til et tegn i sekundet. Overføres et binært tegn per enhed, svarer skridthastigheden til en overførselshastighed på bit/sek.

**BCD-format:**

Tallene 0-9 noteret i total-system. Hvert tal udgøres af 4 cifre (bits).

**BIOS:**

Basic Input Output System.

**Bidirektional:**

Det er muligheden for at sende data i begge retninger, d.v.s. at sende og modtage. Dataledninger fungerer i reglen bidirektionalt, mens adresseledninger kun fungerer unidirektionalt.

**Bloksøgning:**

En gruppe af data søges.

**Bloktransfer:**

En gruppe af data overføres.

**Breakpoints:**

Brudpunkter.

**Brugerinterface:**

Koblingen mellem bruger og datamat.

**Buffer:**

Ved en buffer forståes dels et lagermedie, til kortfristet lagring af information, og dels benyttes udtrykket også for et drev til kobling af større belastninger.

**Bug (lus):**

Programfejl.

**BUS:**

Herunder forståes et ledningssystem til parallel ingormationsoverførsel indenfor computeren, eller mellem flere apparater eller komponenter.

## C.

**Cartridge:**

En modul, der indeholder en ROM-kreds hvori et program er lagret. De fleste hjemmecomputere er forsynet med en såkaldt port, hvori modulet (cartridge) forbindes med computeren. Fordelen ved at bruge en cartridge som lagermedium i stedet for bånd eller diskette, er den ultrakorte tid, det tager at udføre et program fra ROM-kredsen i modulet til computerens arbejdslager. Det er især spil og alternative programmeringssprog, der forhandles i cartridge-form.

**Cluster:**

En samling af sektore.

**Codesegment:**

Kode afsnit.

**Compiler:**

Er betegnelsen for en "oversætter". Et højniveausprog som BASIC skal for det kan "køre" omdannes til maskinsprog som computeren kan forstå. I normal BASIC sker dette løbende efterhånden som programmet afvikles. Denne form for oversættelse kaldes fortolkning, men i stedet for at anvende denne relativt langsomme metode, kan man oversætte sit program een gang for alle. Det er det, man skal bruge en compiler til.

**Converter (forvandler):**

Omvende eller omforme. F.eks. ændre et program, der kan afvikles på en datamat til at kunne afvikles på en anden datamat. Overførelse af data fra et medium til et andet, f.eks. fra hulkort til diskette eller fra mennesker til maskinlæsbar form.

**Cursor:**
Den lille markør på skærmen, der viser hvor "man" befinder sig. Markøren kan have flere former, og kan enten være blinkende eller fast. Cursoren svarer til spidsen af en blyant.

## D.

**Database:**
Samling af data, der er lagret i filer på en sådan måde, at de nemt kan anvendes til andre formål end dem, som var forudset fra begyndelsen.

**Databus**
Se bus.

**Datasegmentet:**
Dataafsnit.

**Datatransmision:**
Data sendes ad elektronisk vej fra et sted til et andet.

**Debug:**
Afluse et program. Finde og rette en fejl i et program eller i sit maskinel.

**Digit:**
Ciffer eller position.

**Diskette:**
En rund magnetiserbar polyesterskive indkapslet i et plasthylster. Ydre lagringsmedie for en computer. Kapaciteten kan variere fra under 100 Kbytes til over 1 Mbytes.

**DMA:**
Direct Memory Access. Dette er et direkte indgreb i datamatens lager, ved hvilket styrringen ikke foregår fra CPU'en, men fra en del af den tilsluttede periferi.

**Drev (in- output):**
Ved et drev forståes enten et program til styring af signaler, der via en in-/outputkanal står i forbindelse med et eksternt apparat, eller en kobling, der bruges til forstærkning af elektriske signaler.

**Duplex:**

En form for dataoverførsel, ved hvilken der findes en ledning for hver overførselsretning, kaldes duplex.

## E.

**Editor:**

I mange programmeringssprog er der en editor, til at redigere i sit program så det virker efter hensigten.

**Eneable:**

Frigivelse, ofte ved IC'er benyttet som chip-enable, menes der hermed frigivelsesindgangen, med hvilken IC'en kan aktiveres.

**Errorlevel:**

Fejlniveau. Hvis der en fejl i et program kommer der en fejlmelding frem på skærmen. Der skrives Error på skærmen efterfulgt af et tal eller en sætning. Vi kender errorlevel fra lommeregneren med det omvendte E.

**Even:**

= lige i betydningen lige tal i modsætning til ulige.

## F.

**Fan-in:**

Indgangsbelastningsfaktor.

**Far-jump:**

Langt-spring i hukommelsen.

**Fil:**

En ordnet samling af sammenhørende data. I daglig tale refereres til den fysiske tilstedeværelse af et EDB-program/dataoplysninger på en diskette eller et bånd.

**Flag:**

Herunder forståes en markering, der har til formål at fastholde en indtruffen tilstand.

**Formattering:**

Den proces, der inddeler en diskette i et antal spor og sektorer for skrivning og læsning af data.

## H.

**Halvduplex:**

Ved dataoverførsel betegnes den driftsmodus som halvduplex, ved hvilken, der nok kan sendes i to retninger, men kun i en retning af gangen.

**Handshaking:**

Dette er en særlig form for dataoverførsel, ved hvilken der ved afsendelse og modtagelse af data, via særlige ledninger, kvitteres for hhv. afsendelse og modtagelse. På den måde bekræftes fx. et "DATA READY" fra afsenderen med et

**Hex-Dump:**

Udskrift af et program i hexadecimalform.

**Hexadecimal:**

Ved det hexadecimal talsystem forståes en fremstillingsform til basis 16. Dette systems cifre går fra 0 til F.

**High order:**

= højeste position.

## I.

**Implementere:**

At udføre det arbejde der kræves for at gøre teori til praksis.

**Indexerede rækker:**

Rækker der har fået index.

**Initialisering:**

De første skridt i et program, med henblik på en defineret starttilstand, betegnes som initialisering.

**Interface:**
Herved forståes et forbindelsesled mellem to enheder (komponenter, kredsløb, programmer, datameter etc.) gennem hvilket udvekslingen af data og/eller styringsinformationer foregår.

**Interrupt:**
Afbrydelse.

**Inversere:**
Omvende.

## K.

**Keyboard:**
Tastatur.

**Kompatible:**
= udskiftelig, kan direkte erstattes af, forenelig med o. lign. To datamater siges at være kompatible, hvis de kan afvikle det samme program uden ændringer, kommunikere med hinanden eller bruge de samme data.

**Konfiguration:**
Betegnelse for indholdet af de enkelte enheder i et samlet computer-anlæg. Enhederne omfatter både hukommelse, interfaces og tilslutningen af printere, diskettestationer, monitorer.

**Konverteres:**
Omvende eller omforme.

**Krympe:**
Formindske.

## L.

**Løkke:**
Programdel, hvis instruktioner skal gentages, indtil en bestemt betingelse er opfyldt.

**Label:**
Identifikation, mærke eller etiket.

**Latch:**
Ved en latch forståes et særligt lager, der i forbindelse med dataudvekslingen mellem datamat og periferi er nødvendigt til tilpasning af de forskellige overførselshastigheder.

**Load:**
Kommando der bevirker at et bestemt program indlæset fra baggrundslager.

**Low order:**
= laveste position.

**M.**

**Markør:**
Se under CURSOR.

**Mark:**
I forbindelse med den serielle dataoverførsel betegnes det logiske niveau "HIGH" som MARK.

**Maskere:**
En betegnelse ved hjælp af hvilken en interrupts gennemførelse kan forhindres.

**Matrix:**
Rektangulær opstilling af elementer i række inden for matematikken.

**Mnemokode:**
Forkortelser som ADD og RET kaldes for Mnemkode.

**N.**

**Near-jump:**
Kort spring.

**Negeres:**
Benægte (gøre negativ).

**Non-nominates:**
Endnu ikke navngivet.

## O.

**Odd:**
= ulige.

**Open Collector:**
Det er en kobling, ved hvilken sluttransistoren har en åben collector, der er ført ud.

**Operand:**
Data, som kan behandles ved hjælp af en operator.

**Overflow (Stakoverløb):**
Datamatens kapacitet (lager) er opbrugt.

## P.

**Parametre:**
En variabel, hvis værdi er fast i en given anvendelse.

**Parity:**
= lighed, paritet.

**Periferi:**
Er et begreb for alle apparater, der kan betjenes fra datamaten uden dog at høre med til denne.

**Pixel:**
Et enkelt element i et billede/tegn.

**Port:**
Dataindgang til centealenheden (CPU). Bruges ofte som betegnelse for alle ud- og indgange på en computer.

**Printer:**
En slags elektrisk skrivemaskine uden tastatur. Computeren kan aktivere printerens skrivehoved, således at de enkelte bogstaver og tegn kan udskrives på papir.

**Processor:**
Del af central enheden i en datamat.

**Promtsymbol:**
Klartegn.

**Pseude-akkumulator:**
Kunstig-akkumulator.

**Pull-up-modstand:**
For at garantere et defineret niveau i inaktiv tilstand, anbringes en sådan modstand ved low-aktive signaler på + forsyningsspændingen.

**PUSH:**
Skubbe.

**R.**

**RAM:**
Random Acces Memory. Er et lager i centralenheden med direkte adgang til.

**RET:**
RET er forkortelse for tilbagespring fra et underprogram.

**Resident:**
Bolig. Et resident program, er et program der ligger i maskinen, men som kun bliver aktivt når en given situation opstår.

**RS232C:**
Det er en seriel dataoverførselsnorm efter den amerikanske EIA-standard op til 19200 baud. Mellemleddet er således defineret, at dataoverførslen foregår asynkront med en starbit før og en eller to stopbits efter hvert tegn. Dette mellemled er ikke TTL-kompatibelt, da spændinger mellem 3 og 15 V anses for et H-niveau og spændinger på mellem -3 og -15 V for et L-niveau. RS232C-mellemleddet kan fungere på afstande på op til 30m.

**Rutiner:**
Program eller dele af et program, der anvendes almindeligtt eller hyppigt.

**S.**

**Scrollning:**
Rulning (skærmbilledet).

**Sekventiel:**
Betyder på hinanden følgende i modsætning til parallel. Udtrykket seriel anvendes synonymt.

**Simulering:**
Efterligning af virkelige forhold ved hjælp af en model.

**Sound-chippen:**
Lyd-chippen.

**Source:**
Kilde.

**Space:**
Ved den serielle dataoverførsel betegnes det logiske niveau "low" som space.

**Stackpointer:**
Stackpointeren er en "symbolsk" viser der peger på et sted i hukommelsen, når man vælger et programpunkt.

**Standardinput:**
Standard-, norm-, målestok-, mønster-indtastning. Indtastning af en bestemt slags data.

**Streng-variabler:**
Navnet på en streng (følge af bogstaver, tal og/eller tegn).

**String (alfanumeriske data):**
Følge af tal, tegn eller bogstaver.

**Synkron:**
Synkron har flere betydninger:
a) synkron = takststyret
b) Ved en synkron dataoverførsel ligger alle binære tegn i faste tidsrater.
c) synkroniseret = fungerende med samme grundtakt

**Syntaks:**
Regler for dannelse af tilladelige konstruktioner i et sprog.

## T.

**Terminate:**

Begrænse, ende, afslutte, slutte, bringe til ophør, ophæve. At terminate et program, vil sige at man afslutter et program.

**Tilslutning (port):**

(port-) udgang eller (port-) indgang er der, hvor kablerne forbinder de enkelte enheder i et datasystem.

**Trace-modus:**

Enkeltskridts-modus. Bruges ved fejlsøgning i programmer.

**3-D-grafik:**

Tre-dimentionalt grafik.

**Tristate drev:**

Dette er et kredsløb, der skelner mellem tre forskellige tilstande: 0, 1 eller udefineret, d.v.s. åben.

**TTL niveau:**

Ved TTL-komponenter defineres et L-niveau ved en spænding på mellem 0 og 0,8 V og et H-niveau ved en spænding på mellem 2,4 og 5 V. Det mellemliggende spændingsområde er udefineret.

## V.

**Valid:**

= gyldig.

**Valid tegn:**

Lovligt tegn.

**Vektor:**

Numerisk variable med en dimension.

**Vektor Interrupt:**

Med en udløst interrupt vil CPU'en udføre en særlig afbrydelsesrutine.

**W.**

**Word-division:**

Ord-deling, ord-inddeling, ord-opdeling, ord-fordeling.

*Ovenstående ordliste er, som angivet i overskriften, ikke komplet, men indholdet er et skønsomt antal af de fremmedord, som optræder i bogen. Skulle der, efter din opfattelse være glemt nogle ord der burde have været med, hører vi gerne herom SKRIFTLIGT.*

EN DATA BECKER BOG
WELTNER & HORNIG
COMMODORE
AMIGA
TIPS OG TRICKS
En guldgrube for AMIGA-folk
DANSK
NORSK
udgave
Forlag: NCS, box 105, DK 6950 Ringkøbing

**248,-**

EN DATA BECKER BOG
SCHUMANN
COMMODORE
AMIGA
& ATARI m. fl.
C for begyndere m. fl.
Sproget C er AMIGA-venligt
DANSK
udgave
NORSK
Forlag: NCS, box 105, DK 6950 Ringkøbing

**248,-**

# ASSEMBLER

## AMIGA PROFIMAT

Den komplette programpakke for enhver Amiga Maskinsprogsprogrammør.
Extrem hurtig, er skrevet i Assembler.

- **Integreret Editor, Debugger, Disassembler og Reassembler.**
- **Omfangsrigt System-bibliotek.**
- **Kører under CLI & Workbench.**
- **Relativ og absolut Code.**
- **Fejlsøgningsfunktioner.**
- **Cross-Reference-Liste.**
- **Fuld 32 Bits aritmetik.**
- **Debugger med 68020-Singel-Step-Emulation.**
- **Kører på Amiga med 512 Kb og Kickstart 1.2.**

**Kun kr. 498,00**
inkl. moms

NORDIC COMPUTER SOFTWARE
SMEDEGADE 7
POSTBOX 105 . DK-6950 RINGKØBING

# IC RUN

***Tegn abonnement på IC RUN idag og du får helt gratis to fuldprisspil på bånd til en værdi af over 300 kroner samlet.***

***Det drejer sig om et af verdens absolut bedste spil til Commodore 64: BOULDER DASH CONSTRUCTION SET, en anmelderfavorit, der har solgt i over 300.000 eksemplarer.***

***Ved siden af får du »Indoor SPORTS«, også et Commodore 64-spil, der er så godt at en AMIGA-version snart er på trapperne.***

★ For de seriøse med diskstation, kan der istedet vælges et praktisk tekstbehandlingsprogram, nemlig BWW Tekst 1 til Commodore 64, der har en værdi af kr. 398,-.

★ **AMIGA GOLD er IC RUN's nyeste tilbud til Amiga ejere. En diskette fyldt med nogle af de mest interessante Public Domain og ShareWare programmer på markedet. AMIGA GOLD indeholder over 25 programmer, lydeksempler og DPaint billeder, der virkelig viser hvad Amigaen duer til.**

- ring til vor abonnements afdeling: 02 54 25 24

ISBN 87-7283-023-9

This was brought to you

from the archives of

http://retro-commodore.eu